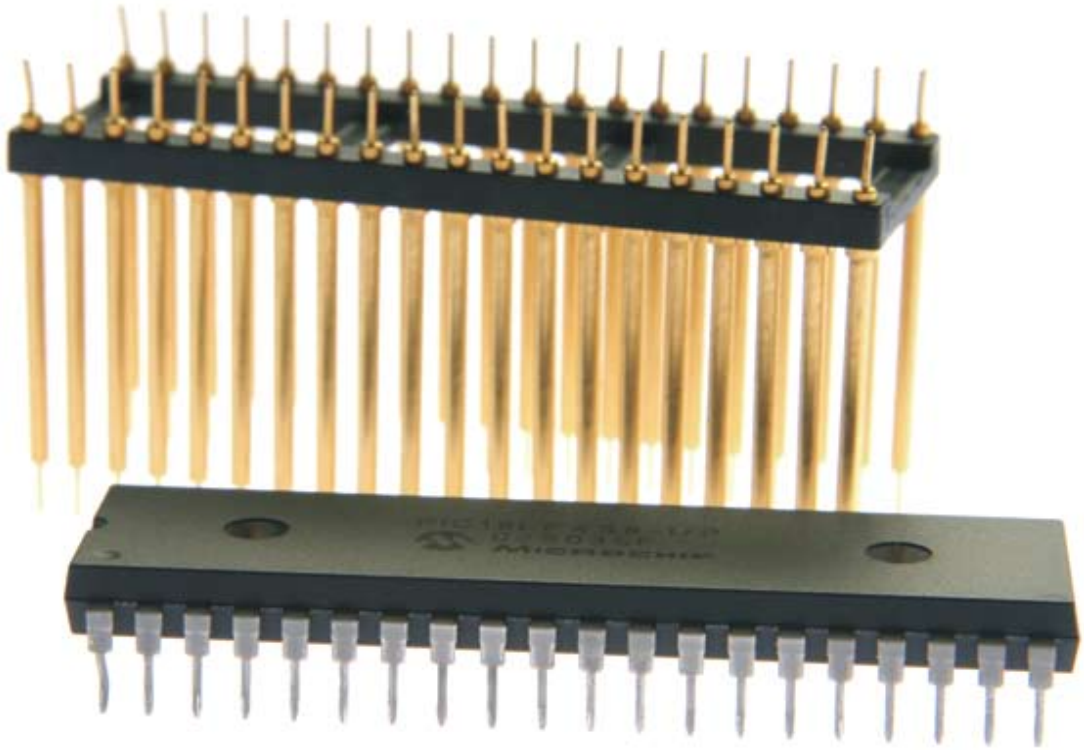
HI-TECH SOFTWARE

Manual

HI-TECH C® Tools for the PIC18 MCU Family



HI-TECH C Compiler for PIC18 MCUs

Microchip Technology Inc.

Copyright (C) 2011 Microchip Technology Inc.
All Rights Reserved. Printed in Australia.
Produced on: September 27, 2011

Australian Design Centre
45 Colebard Street West
Acacia Ridge QLD 4110
Australia

web: http://www.microchip.com

Contents

Table of Contents **3**

List of Tables **17**

1 Introduction **19**
1.1 Typographic conventions . . . 19

2 PICC18 Command-line Driver **21**
2.1 Invoking the Compiler . . . 22
2.1.1 Long Command Lines . . . 23
2.2 The Compilation Sequence . . . 24
2.2.1 Single-step Compilation . . . 25
2.2.2 Generating Intermediate Files . . . 26
2.2.3 Special Processing . . . 27
2.2.3.1 Printf check . . . 28
2.2.3.2 Assembly Code Requirements . . . 28
2.3 Runtime Files . . . 28
2.3.1 Library Files . . . 29
2.3.1.1 Standard Libraries . . . 30
2.3.1.2 Utility Libraries . . . 30
2.3.1.3 Peripheral Libraries . . . 30
2.3.2 Runtime Startup Code . . . 31
2.3.2.1 Initialization of Data psects . . . 32
2.3.2.2 Clearing the Bss Psects . . . 32
2.3.3 The Powerup Routine . . . 33
2.3.4 The `printf` Routine . . . 34
2.4 Debugging Information . . . 35
2.5 Compiler Messages . . . 36

2.5.1 Messaging Overview . . . 36
2.5.2 Message Language . . . 37
2.5.3 Message Type . . . 37
2.5.4 Message Format . . . 38
2.5.5 Changing Message Behaviour . . . 40
2.5.5.1 Disabling Messages . . . 40
2.5.5.2 Changing Message Types . . . 41
2.6 PICC18 Driver Option Descriptions . . . 41
2.6.1 Option Formats . . . 41
2.6.2 -C: Compile to Object File . . . 42
2.6.3 -D*macro*: Define Macro . . . 42
2.6.4 -E*file*: Redirect Compiler Errors to a File . . . 43
2.6.5 -G*file*: Generate Source-level Symbol File . . . 43
2.6.6 -I*path*: Include Search Path . . . 44
2.6.7 -L*library*: Scan Library . . . 44
2.6.8 -L-*option*: Adjust Linker Options Directly . . . 45
2.6.9 -M*file*: Generate Map File . . . 47
2.6.10 -N*size*: Identifier Length . . . 47
2.6.11 -O*file*: Specify Output File . . . 47
2.6.12 -P: Preprocess Assembly Files . . . 48
2.6.13 -Q: Quiet Mode . . . 48
2.6.14 -S: Compile to Assembler Code . . . 48
2.6.15 -U*macro*: Undefine a Macro . . . 48
2.6.16 -V: Verbose Compile . . . 49
2.6.17 -X: Strip Local Symbols . . . 49
2.6.18 --ADDRQUAL: Set Compiler Response to Memory Qualifier . . . 49
2.6.19 --ASMLIST: Generate Assembler .LST Files . . . 49
2.6.20 --CHECKSUM=*start-end@destination<,specs>*: Calculate a checksum . . . 50
2.6.21 --CHIP=*processor*: Define Processor . . . 50
2.6.22 --CHIPINFO: Display List of Supported Processors . . . 50
2.6.23 --CMODE: Specify compatibility mode . . . 51
2.6.24 --CODEOFFSET: Offset Program Code to Address . . . 51
2.6.25 --CR=*file*: Generate Cross Reference Listing . . . 51
2.6.26 --DEBUGGER=*type*: Select Debugger Type . . . 52
2.6.27 --DOUBLE=*type*: Select kind of Double Types . . . 52
2.6.28 --ECHO: Echo command line before processing . . . 52
2.6.29 --EMI=*type*: Select operating mode of the external memory interface (EMI) 52
2.6.30 --ERRATA=*type*: Specify to add or remove specific errata workarounds . . . 53

2.6.31 --ERRFORMAT=*format*: Define Format for Compiler Messages 53
2.6.32 --ERRORS=*number*: Maximum Number of Errors 53
2.6.33 --FILL=*opcode*: Fill Unused Program Memory 53
2.6.34 --FLOAT=*type*: Select kind of Float Types 54
2.6.35 --GETOPTION=*app*,*file*: Get Command-line Options 54
2.6.36 --HELP<=*option*>: Display Help 54
2.6.37 --HTML: Generate HTML Debug Files 54
2.6.38 --IDE=*type*: Specify the IDE being used 55
2.6.39 --LANG=*language*: Specify the Language for Messages 55
2.6.40 --MEMMAP=*file*: Display Memory Map 55
2.6.41 --MODE=*mode*: Choose Compiler Operating Mode 56
2.6.42 --MSGDISABLE=*messagelist*: Disable Warning Messages 56
2.6.43 --MSGFORMAT=*format*: Set Advisory Message Format 56
2.6.44 --NODEL: Do not Remove Temporary Files 56
2.6.45 --NOEXEC: Don't Execute Compiler 56
2.6.46 --OBJDIR=*dir*: Specify a Directory for Intermediate Files 57
2.6.47 --OPT<=*type*>: Invoke Compiler Optimizations 57
2.6.48 --OUTDIR=*path*: Specify a Directory for Output Files 57
2.6.49 --OUTPUT=*type*: Specify Output File Type 58
2.6.50 --PASS1: Compile to P-code 58
2.6.51 --PRE: Produce Preprocessed Source Code 58
2.6.52 --PROTO: Generate Prototypes 59
2.6.53 --RAM=*lo-hi*,<*lo-hi*,...>: Specify Additional RAM Ranges 60
2.6.54 --ROM=*lo-hi*,<*lo-hi*,...>|*tag*: Specify Additional ROM Ranges . . . 61
2.6.55 --RUNTIME=*type*: Specify Runtime Environment 63
2.6.56 --SCANDEP: Scan for Dependencies 63
2.6.57 --SERIAL=*hexcode@address*: Store a Value at this Program Memory Address 63
2.6.58 --SETOPTION=*app*,*file*: Set The Command-line Options for Application 63
2.6.59 --SHROUD: Obfuscate p-code Files 64
2.6.60 --STRICT: Strict ANSI Conformance 64
2.6.61 --SUMMARY=*type*: Select Memory Summary Output Type 64
2.6.62 --TIME: Report time taken for each phase of build process 64
2.6.63 --VER: Display The Compiler's Version Information 65
2.6.64 --WARN=*level*: Set Warning Level 65
2.6.65 --WARNFORMAT=*format*: Set Warning Message Format 65
2.7 MPLAB IDE v8 Universal Toolsuite Equivalents 66
2.7.1 Directories Tab 66
2.7.2 Compiler Tab 66

2.7.3 Linker Tab . . . 69
2.7.4 Global Tab . . . 72
2.8 MPLAB X Universal Toolsuite Equivalents . . . 73
2.8.1 Compiler Category . . . 74
2.8.1.1 Messages . . . 74
2.8.1.2 Address Qualifiers . . . 74
2.8.1.3 Operation . . . 75
2.8.1.4 Preprocessor . . . 75
2.8.1.5 Optimization . . . 76
2.8.2 Linker Category . . . 77
2.8.2.1 Data . . . 77
2.8.2.2 Report . . . 78
2.8.2.3 Runtime . . . 79
2.8.2.4 Code . . . 79
2.8.2.5 Additional . . . 79

3 C Language Features **83**
3.1 ANSI Standard Issues . . . 83
3.1.1 Divergence from the ANSI C Standard . . . 83
3.1.2 Implementation-defined behaviour . . . 83
3.1.3 Non-ANSI Operations . . . 84
3.1.4 C18 Compatibility . . . 84
3.2 Processor-related Features . . . 85
3.2.1 Processor Support . . . 85
3.2.2 Device Header Files . . . 86
3.2.3 Stack . . . 86
3.2.4 Configuration Fuses . . . 86
3.2.5 ID Locations . . . 88
3.2.6 Bit Instructions . . . 88
3.2.7 EEPROM and Flash Runtime Access . . . 89
3.2.7.1 EEPROM Access . . . 89
3.2.7.2 Flash Access . . . 90
3.2.8 Using SFRs From C Code . . . 91
3.2.8.1 Multi-byte SFRs . . . 92
3.3 Supported Data Types and Variables . . . 93
3.3.1 Radix Specifiers and Constants . . . 93
3.3.2 Bit Data Types and Variables . . . 95
3.3.3 Using Bit-Addressable Registers . . . 96
3.3.4 8-Bit Integer Data Types and Variables . . . 96

3.3.5 16-Bit Integer Data Types . . . 96
3.3.6 24-Bit Integer Data Types . . . 97
3.3.7 32-Bit Integer Data Types and Variables . . . 97
3.3.8 Floating Point Types and Variables . . . 98
3.3.9 Structures and Unions . . . 99
3.3.9.1 Bit-fields in Structures . . . 99
3.3.9.2 Structure and Union Qualifiers . . . 100
3.3.10 Standard Type Qualifiers . . . 101
3.3.10.1 Const and Volatile Type Qualifiers . . . 101
3.3.11 Special Type Qualifiers . . . 102
3.3.11.1 Persistent Type Qualifier . . . 102
3.3.11.2 Near Type Qualifier . . . 103
3.3.11.3 Far Type Qualifier . . . 103
3.3.12 Pointer Types . . . 104
3.3.12.1 Combining Type Qualifiers and Pointers . . . 104
3.3.12.2 Data Pointers . . . 106
3.3.12.3 Pointers to Const . . . 107
3.3.12.4 Pointers to Both Memory Spaces . . . 108
3.3.12.5 Function Pointers . . . 109
3.4 Storage Class and Object Placement . . . 110
3.4.1 Local Variables . . . 110
3.4.1.1 Auto Variables . . . 110
3.4.1.2 Static Variables . . . 114
3.4.2 Absolute Variables . . . 115
3.4.2.1 Absolute Variables in Data Memory . . . 115
3.4.2.2 Absolute Variables in Program Memory . . . 115
3.4.3 Objects in Program Space . . . 116
3.4.4 Dynamic Memory Allocation . . . 116
3.4.5 Memory Models . . . 116
3.5 Functions . . . 116
3.5.1 Absolute Functions . . . 116
3.5.2 External Functions . . . 117
3.5.3 Function Argument Passing . . . 117
3.5.4 Function Return Values . . . 119
3.5.4.1 Structure Return Values . . . 119
3.6 Operators . . . 119
3.6.1 Integral Promotion . . . 119
3.6.2 Shifts applied to integral types . . . 121
3.6.3 Division and modulus with integral types . . . 121

3.7 Register Usage . . . 122
3.8 Psects . . . 123
3.8.1 Compiler-generated Psects . . . 123
3.8.1.1 Program Space Psects . . . 124
3.8.1.2 Data Space Psects . . . 125
3.9 Interrupt Handling in C . . . 126
3.9.1 Interrupt Functions . . . 126
3.9.2 Context Switching . . . 127
3.9.2.1 Context Saving . . . 127
3.9.2.2 Context Retrieval . . . 128
3.9.3 Enabling Interrupts . . . 128
3.9.4 Function Duplication . . . 128
3.9.4.1 Disabling Duplication . . . 129
3.9.5 Interrupt Registers . . . 130
3.10 Mixing C and Assembly Code . . . 131
3.10.1 External Assembly Language Functions . . . 131
3.10.2 #asm, #endasm and asm() . . . 134
3.10.3 Accessing C objects from within Assembly Code . . . 135
3.10.3.1 Accessing special function register names from assembler . . . 136
3.10.4 Interaction between Assembly and C Code . . . 138
3.10.4.1 Absolute Psects . . . 138
3.10.4.2 Undefined Symbols . . . 139
3.11 Preprocessing . . . 139
3.11.1 C Language Comments . . . 140
3.11.2 Preprocessor Directives . . . 140
3.11.3 Predefined Macros . . . 140
3.11.4 Pragma Directives . . . 143
3.11.4.1 The #pragma printf_check Directive . . . 143
3.11.4.2 The #pragma regsused Directive . . . 145
3.11.4.3 The #pragma switch Directive . . . 146
3.11.4.4 The #pragma inline Directive . . . 147
3.11.4.5 The #pragma interrupt_level Directive . . . 147
3.11.4.6 The #pragma warning Directive . . . 147
3.12 Linking Programs . . . 149
3.12.1 Replacing Library Modules . . . 150
3.12.2 Signature Checking . . . 150
3.12.3 Linker-Defined Symbols . . . 152
3.13 Standard I/O Functions and Serial I/O . . . 152

4 Macro Assembler **153**
4.1 Assembler Usage . . . 153
4.2 Assembler Options . . . 154
4.3 HI-TECH C Assembly Language . . . 157
4.3.1 Assembler Format Deviations . . . 157
4.3.2 Pre-defined Macros . . . 158
4.3.3 Statement Formats . . . 158
4.3.4 Characters . . . 158
4.3.4.1 Delimiters . . . 159
4.3.4.2 Special Characters . . . 159
4.3.5 Comments . . . 159
4.3.5.1 Special Comment Strings . . . 159
4.3.6 Constants . . . 160
4.3.6.1 Numeric Constants . . . 160
4.3.6.2 Character Constants and Strings . . . 160
4.3.7 Identifiers . . . 160
4.3.7.1 Significance of Identifiers . . . 161
4.3.7.2 Assembler-Generated Identifiers . . . 161
4.3.7.3 Location Counter . . . 161
4.3.7.4 Register Symbols . . . 162
4.3.7.5 Symbolic Labels . . . 162
4.3.8 Expressions . . . 162
4.3.9 Program Sections . . . 164
4.3.10 Assembler Directives . . . 165
4.3.10.1 GLOBAL . . . 165
4.3.10.2 END . . . 165
4.3.10.3 PSECT . . . 167
4.3.10.4 ORG . . . 169
4.3.10.5 EQU . . . 169
4.3.10.6 SET . . . 169
4.3.10.7 DB . . . 170
4.3.10.8 DW . . . 170
4.3.10.9 DS . . . 170
4.3.10.10 DABS . . . 170
4.3.10.11 FNCALL . . . 171
4.3.10.12 FNROOT . . . 171
4.3.10.13 IF, ELSIF, ELSE and ENDIF . . . 171
4.3.10.14 MACRO and ENDM . . . 172
4.3.10.15 LOCAL . . . 173

4.3.10.16 ALIGN . . . 173
4.3.10.17 REPT . . . 174
4.3.10.18 IRP and IRPC . . . 174
4.3.10.19 BANKSEL . . . 175
4.3.10.20 PROCESSOR . . . 175
4.3.10.21 SIGNAT . . . 176
4.3.11 Assembler Controls . . . 176
4.3.11.1 ASMOPT_OFF and ASMOPT_ON . . . 176
4.3.11.2 COND . . . 176
4.3.11.3 EXPAND . . . 176
4.3.11.4 INCLUDE . . . 178
4.3.11.5 LIST . . . 178
4.3.11.6 NOCOND . . . 178
4.3.11.7 NOEXPAND . . . 178
4.3.11.8 NOLIST . . . 179
4.3.11.9 NOXREF . . . 179
4.3.11.10 PAGE . . . 179
4.3.11.11 STACK . . . 179
4.3.11.12 SUBTITLE . . . 179
4.3.11.13 TITLE . . . 179
4.3.11.14 XREF . . . 179
4.4 Assembly List Files . . . 180
4.4.1 General Format . . . 180
4.4.2 Function Information . . . 181
4.4.3 Pointer Reference Graph . . . 181
4.4.4 Call Graph . . . 183
4.4.5 Call Graph Critical Paths . . . 185

5 Linker and Utilities **187**
5.1 Introduction . . . 187
5.2 Relocation and Psects . . . 187
5.3 Program Sections . . . 188
5.4 Local Psects . . . 188
5.5 Global Symbols . . . 188
5.6 Link and load addresses . . . 189
5.7 Operation . . . 189
5.7.1 Numbers in linker options . . . 190
5.7.2 -A*class=low-high,...* . . . 191
5.7.3 -C*x* . . . 191

5.7.4 -C*psect=class* . . . 191
5.7.5 -D*class=delta* . . . 191
5.7.6 -D*symfile* . . . 191
5.7.7 -E*errfile* . . . 192
5.7.8 -F . . . 192
5.7.9 -G*spec* . . . 192
5.7.10 -H*symfile* . . . 193
5.7.11 -H+*symfile* . . . 193
5.7.12 -J*errcount* . . . 193
5.7.13 -K . . . 193
5.7.14 -I . . . 193
5.7.15 -L . . . 193
5.7.16 -LM . . . 194
5.7.17 -M*mapfile* . . . 194
5.7.18 -N, -Ns and-Nc . . . 194
5.7.19 -O*outfile* . . . 194
5.7.20 -P*spec* . . . 194
5.7.21 -Q*processor* . . . 196
5.7.22 -S . . . 196
5.7.23 -S*class=limit[, bound]* . . . 196
5.7.24 -U*symbol* . . . 196
5.7.25 -V*avmap* . . . 197
5.7.26 -W*num* . . . 197
5.7.27 -X . . . 197
5.7.28 -Z . . . 197
5.8 Invoking the Linker . . . 197
5.9 Map Files . . . 198
5.9.1 Generation . . . 198
5.9.2 Contents . . . 198
5.9.2.1 General Information . . . 199
5.9.2.2 Psect Information listed by Module . . . 200
5.9.2.3 Psect Information listed by Class . . . 201
5.9.2.4 Segment Listing . . . 202
5.9.2.5 Unused Address Ranges . . . 202
5.9.2.6 Symbol Table . . . 203
5.10 Librarian . . . 204
5.10.1 The Library Format . . . 204
5.10.2 Using the Librarian . . . 204
5.10.3 Examples . . . 205

5.10.4 Supplying Arguments . . . 206
5.10.5 Listing Format . . . 206
5.10.6 Ordering of Libraries . . . 206
5.10.7 Error Messages . . . 207
5.11 Objtohex . . . 207
5.11.1 Checksum Specifications . . . 207
5.12 Cref . . . 209
5.12.1 -F*prefix* . . . 209
5.12.2 -H*heading* . . . 210
5.12.3 -L*len* . . . 210
5.12.4 -O*outfile* . . . 210
5.12.5 -P*width* . . . 210
5.12.6 -S*stoplist* . . . 210
5.12.7 -X*prefix* . . . 210
5.13 Cromwell . . . 211
5.13.1 -Pname[,architecture] . . . 211
5.13.2 -N . . . 211
5.13.3 -D . . . 213
5.13.4 -C . . . 213
5.13.5 -F . . . 213
5.13.6 -Okey . . . 213
5.13.7 -Ikey . . . 213
5.13.8 -L . . . 213
5.13.9 -E . . . 213
5.13.10 -B . . . 214
5.13.11 -M . . . 214
5.13.12 -V . . . 214
5.14 Hexmate . . . 214
5.14.1 Hexmate Command Line Options . . . 215
5.14.1.1 specifications,filename.hex . . . 215
5.14.1.2 + Prefix . . . 217
5.14.1.3 -ADDRESSING . . . 217
5.14.1.4 -BREAK . . . 217
5.14.1.5 -CK . . . 218
5.14.1.6 -FILL . . . 218
5.14.1.7 -FIND . . . 220
5.14.1.8 -FIND...,DELETE . . . 221
5.14.1.9 -FIND...,REPLACE . . . 221
5.14.1.10 -FORMAT . . . 221

5.14.1.11 -HELP . . . 222
5.14.1.12 -LOGFILE . . . 222
5.14.1.13 -MASK . . . 223
5.14.1.14 -O*file* . . . 223
5.14.1.15 -SERIAL . . . 223
5.14.1.16 -SIZE . . . 224
5.14.1.17 -STRING . . . 224
5.14.1.18 -STRPACK . . . 225

A Library Functions **227**
__CONFIG . . . 228
__EEPROM_DATA . . . 229
__IDLOC . . . 230
_DELAY . . . 231
_DELAY3 . . . 232
ABS . . . 233
ACOS . . . 234
ASCTIME . . . 235
ASIN . . . 237
ASSERT . . . 238
ATAN . . . 239
ATAN2 . . . 240
ATOF . . . 241
ATOI . . . 242
ATOL . . . 243
BSEARCH . . . 244
CEIL . . . 246
CGETS . . . 247
CLRWDT . . . 249
CONFIG_READ . . . 250
COS . . . 252
COSH . . . 253
CPUTS . . . 254
CTIME . . . 255
device_id_read . . . 256
DI . . . 258
DIV . . . 260
EEPROM_READ . . . 261
EVAL_POLY . . . 263

EXP 264
FABS 265
FLASH 266
FMOD 267
FLOOR 268
FREXP 269
GETCH 270
GETCHAR 271
GETS 272
GMTIME 273
IDLOC_READ 275
ISALNUM 277
ISDIG 279
ITOA 280
LABS 281
LDEXP 282
LDIV 283
LOCALTIME 284
LOG 286
LONGJMP 287
LTOA 289
MEMCMP 290
MEMMOVE 292
MKTIME 293
MODF 295
NOP 296
OS_TSLEEP 297
POW 298
PRINTF 299
PUTCH 302
PUTCHAR 303
PUTS 305
QSORT 306
RAND 308
READTIMER 310
RESET 311
ROUND 312
SETJMP 315
SIN 317

SLEEP . . . 318
SQRT . . . 319
SRAND . . . 320
STRCAT . . . 321
STRCHR . . . 322
STRCMP . . . 324
STRCPY . . . 326
STRCSPN . . . 327
STRLEN . . . 328
STRNCAT . . . 329
STRNCMP . . . 331
STRNCPY . . . 333
STRPBRK . . . 335
STRRCHR . . . 336
STRSPN . . . 337
STRSTR . . . 338
STRTOD . . . 339
STRTOL . . . 341
STRTOK . . . 343
TAN . . . 345
TIME . . . 346
TOLOWER . . . 348
TRUNC . . . 349
UDIV . . . 350
ULDIV . . . 351
UNGETCH . . . 352
UTOA . . . 353
VA_START . . . 354
WRITETIMER . . . 356
XTOI . . . 357

B Error and Warning Messages **359**
1... . . . 359
138... . . . 367
184... . . . 373
226... . . . 381
268... . . . 390
311... . . . 396
354... . . . 404

398... 413
443... 418
487... 426
595... 433
668... 437
720... 442
764... 451
817... 456
866... 462
923... 467
982... 472
1039... 477
1185... 482
1234... 487
1289... 493
1350... 498
0... 505

C Chip Information **507**

Index **515**

List of Tables

2.1 PICC18 input file types . . . 22
2.2 Support languages . . . 37
2.3 Messaging environment variables . . . 38
2.4 Messaging placeholders . . . 39
2.5 Compiler Responses to Memory Qualifiers . . . 49
2.6 Compatibility modes . . . 51
2.7 Supported Double Types . . . 52
2.8 Supported Float Types . . . 54
2.9 Supported IDEs . . . 55
2.10 Supported languages . . . 55
2.11 Optimization Options . . . 57
2.12 Output file formats . . . 58
2.13 Runtime environment suboptions . . . 62
2.14 Memory Summary Suboptions . . . 65

3.1 Basic data types . . . 93
3.2 Radix formats . . . 94
3.3 Floating-point formats . . . 98
3.4 Floating-point format example IEEE 754 . . . 98
3.5 Integral division . . . 121
3.6 Registers Used by the Compiler . . . 122
3.7 Preprocessor directives . . . 144
3.9 Pragma directives . . . 145
3.10 Valid register names . . . 146
3.11 Switch types . . . 146
3.12 Supported standard I/O functions . . . 152

4.1 ASPIC18 command-line options . . . 154
4.2 ASPIC18 statement formats . . . 159
4.3 ASPIC18 numbers and bases . . . 160
4.4 ASPIC18 operators . . . 163
4.5 ASPIC18 assembler directives . . . 166
4.6 PSECT flags . . . 167
4.7 PIC18 assembler controls . . . 177
4.8 LIST control options . . . 178

5.1 Linker command-line options . . . 189
5.1 Linker command-line options . . . 190
5.2 Librarian command-line options . . . 205
5.3 Librarian key letter commands . . . 205
5.4 OBJTOHEX command-line options . . . 208
5.5 CREF command-line options . . . 209
5.6 CROMWELL format types . . . 211
5.7 CROMWELL command-line options . . . 212
5.8 -P option architecture arguments for COFF file output. . . . 212
5.9 Hexmate command-line options . . . 216
5.10 Hexmate Checksum Algorithm Selection . . . 219
5.11 INHX types used in -FORMAT option . . . 222

C.1 Devices supported by HI-TECH C Compiler for PIC18 MCUs . . . 507
C.1 Devices supported by HI-TECH C Compiler for PIC18 MCUs . . . 508
C.1 Devices supported by HI-TECH C Compiler for PIC18 MCUs . . . 509
C.1 Devices supported by HI-TECH C Compiler for PIC18 MCUs . . . 510
C.1 Devices supported by HI-TECH C Compiler for PIC18 MCUs . . . 511
C.1 Devices supported by HI-TECH C Compiler for PIC18 MCUs . . . 512
C.1 Devices supported by HI-TECH C Compiler for PIC18 MCUs . . . 513

Chapter 1

Introduction

1.1 Typographic conventions

Different fonts and styles are used throughout this manual to indicate special words or text. Computer prompts, responses and filenames will be printed in `constant-spaced` type**.** When the filename is the name of a standard header file, the name will be enclosed in angle brackets, e.g. `<stdio.h>`. These header files can be found in the `INCLUDE` directory of your distribution.

Samples of code, C keywords or types, assembler instructions and labels will also be printed in a `constant-space` type. Assembler code is printed in a font similar to that used by C code.

Particularly useful points and new terms will be emphasized using *italicized type*. When part of a term requires substitution, that part should be printed in the appropriate font, but in *italics*. For example: `#include <`*`filename.h`*`>`.

Chapter 2

PICC18 Command-line Driver

`PICC18` is the driver invoked from the command line to perform all aspects of compilation, including C code generation, assembly and link steps. It is the recommended way to use the compiler as it hides the complexity of all the internal applications used in the compilation process and provides a consistent interface for all compilation steps.

This chapter describes the steps the driver takes during compilation, files that the driver can accept and produce, as well as the command-line options that control the compiler's operation.

WHAT IS "THE COMPILER"? Throughout this manual, the term "the compiler" is used to refer to either all, or some subset of, the collection of applications that form the HI-TECH C Compiler for PIC18 MCUs package. Often it is not important to know, for example, whether an action is performed by the parser or code generator application, and it is sufficient to say it was performed by "the compiler".

It is also reasonable for "the compiler" to refer to the command-line driver (or just "driver"), PICC18, as this is the application executed to invoke the compilation process. Following this view, "compiler options" should be considered command-line driver options, unless otherwise specified in this manual.

Similarly "compilation" refers to all, or some part of, the steps involved in generating source code into an executable binary image.

Table 2.1: PICC18 input file types

File Type	Meaning
`.c`	C source file
`.p1`	p-code file
`.lpp`	p-code library file
`.as`	Assembler source file
`.obj`	Relocatable object code file
`.lib`	Relocatable object library file
`.hex`	Intel HEX file

2.1 Invoking the Compiler

This chapter looks at how to use PICC18 as well as the tasks that it and the internal applications perform during compilation.

`PICC18` has the following basic command format:

```
PICC18 [options] files [libraries]
```

It is conventional to supply *`options`* (identified by a leading *dash* "-" or *double dash* "–") before the filenames, although this is not mandatory.

The formats of the options are discussed below in Section 2.6, and a detailed description of each option follows.

The *`files`* may be any mixture of C and assembler source files, and precompiled intermediate files, such as relocatable object (`.obj`) files or p-code (`.p1`) files. The order of the files is not important, except that it may affect the order in which code or data appears in memory, and may affect the name of some of the output files.

`Libraries` is a list of either object code or p-code library files that will be searched by the linker. The `-L` option, see Section 2.6.7, can also be used to specify library files to search.

`PICC18` distinguishes source files, intermediate files and library files solely by the *file type* or *extension*. Recognized file types are listed in Table 2.1. This means, for example, that an assembler file must always have a `.as` extension. Alphabetic case of the extension is not important from the compiler's point of view.

MODULES AND SOURCE FILES: A C *source file* is a file on disk that contains all or part of a program. C source files are initially passed to the preprocessor by the driver. A *module* is the output of the preprocessor, for a given source file, after inclusion of any header files (or other source files) which are specified by `#include` preprocessor

directives. These modules are then passed to the remainder of the compiler applications. Thus, a module may consist of several source and header files. A module is also often referred to as a *translation unit*. These terms can also be applied to assembly files, as they too can include other header and source files.

Some of the compiler's output files contain project-wide information and are not directly associated with any one particular input file, e.g. the map file. If the names of these project-wide files are not specified on the command line, the basename of these files is derived from the first C source file listed on the command line. If there are no files of this type being compiled, the name is based on the first input file (regardless of type) on the command line. Throughout this manual, the basename of this file will be called the *project name*.

Most IDEs use project files whose names are user-specified. Typically the names of project-wide files, such as map files, are named after the project, however check the manual for the IDE you are using for more details.

2.1.1 Long Command Lines

The PICC18 driver is capable of processing command lines exceeding any operating system limitation. To do this, the driver may be passed options via a command file. The command file is read by using the `@` symbol which should be immediately followed (i.e. no intermediate space character) by the name of the file containing the command line arguments.

The file may contain blank lines, which are simply skipped by the driver. The command-line arguments may be placed over several lines by using a *space* and *backslash* character for all non-blank lines, except for the last line.

The use of a command file means that compiler options and project filenames can be stored along with the project, making them more easily accessible and permanently recorded for future use.

TUTORIAL

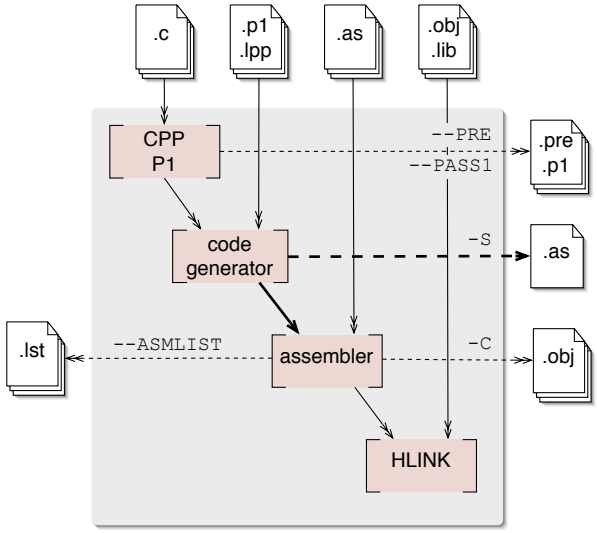
USING COMMAND FILES A command file `xyz.cmd` is constructed with your favorite text editor and contains both the options and file names that are required to compile your project as follows:

```
--chip=18F242 -m \
--opt=all -g \
main.c isr.c
```

After it is saved, the compiler may be invoked with the command:

```
PICC18 @xyz.cmd
```

Figure 2.1: Flow diagram of the initial compilation sequence



2.2 The Compilation Sequence

`PICC18` will check each file argument and perform appropriate actions on each file. The entire compilation sequence can be thought of as the initial sequence up to the link stage, and the final sequence which takes in the link step and any post link steps required.

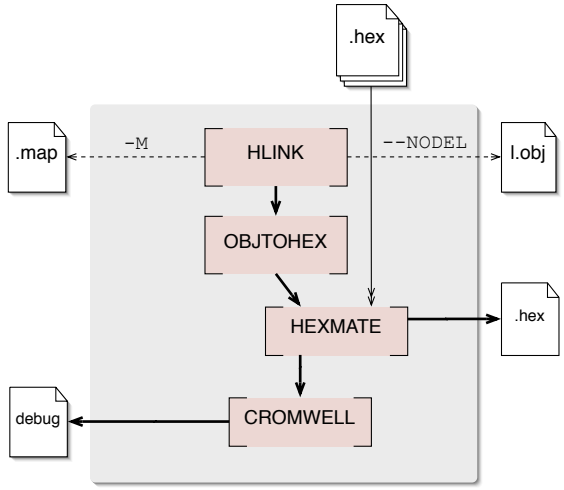
Graphically the compilation steps up to the link stage are illustrated in Figure 2.1. This diagram shows all possible input files along the top; intermediate and transitional files, along the right side; and useful compiler output files along the left. Generated files are shown along with the options that are used to generate and preserve these. All the files shown on the right, can be generated and fed to the compiler in a subsequent compile step; those on the left are used for debug purposes and cannot be used as an input to any subsequent compilation.

The individual compiler applications are shown as boxes. The C preprocessor, `CPP`, and parser, `P1`, have been grouped together for clarity.

The thin, multi-arrowed lines indicate the flow of multiple files — one for each file being processed by the revel ant application. The thick single-arrowed lines indicate a single file for the project being compiled. Thus, for example, when using the `--PASS1` driver option, the parser produces one `.p1` file for each C source file that is being compiled as part of the project, but the code generator produces only one `.as` file from all `.c`, `.p1` and `.lpp` input files which it is passed.

Dotted lines indicate a process that may require an option to create or preserve the indicated file.

Figure 2.2: Flow diagram of the final compilation sequence



The link and post-link steps are graphically illustrated in Figure 2.2.

This diagram shows `.hex` files as additional input file type not considered in the initial compilation sequence. These files can be merged into the `.hex` file generated from the other input files in the project by an application called `HEXMATE`. See Section 5.14 for more information on this utility.

The output of the linker is a single absolute object file, called `l.obj`, that can be preserved by using the `--NODEL` driver option. Without this option, this temporary file is used to generate an output file (e.g. a HEX file) and files used for debugging by development tools (e.g. COFF files) before it is deleted. The file `l.obj` can be used as the input to `OBJTOHEX` if running this application manually, but it cannot be passed to the driver as an input file as it absolute and cannot be further processed.

2.2.1 Single-step Compilation

The command-line driver, PICC18, can compile any mix of input files in a single step. All source files will be re-compiled regardless of whether they have been changes since that last time a compilation was performed.

Unless otherwise specified, a default output file and debug file are produced. All intermediate files (`.p1` and `.obj`) remain after compilation has completed, but all other transitional files are deleted, unless you use the `--NODEL` option which preserves all generated files. Note some generated files may be in a temporary directory not associated with your project and use a pseudo-randomly

generated filename.

TUTORIAL

SINGLE STEP COMPILATION The files, `main.c`, `io.c`, `mdef.as`, `sprt.obj`, `a_sb.lib` and `c_sb.lpp` are to be compiled. To perform this in a single step, the following command line can be used as a starting point for the project development.

```
PICC18 --chip=18F242 main.c io.c mdef.as sprt.obj a_sb.lib c_sb.lpp
```

This will run the C pre-processor then the parser with `main.c` as input, and then again for `io.c` producing two p-code files. These two files, in addition to the library file `c_sb.lpp`, are passed to the code generator producing a single temporary assembler file output. The assembler is then executed and is passed the output of the code generator. It is run again with `mdef.as`, producing two relocatable object files. The linker is then executed, passing in the assembler output files in addition to `sprt.obj` and the library file `a_sb.lib`. The output is a single absolute object file, `l.obj`. This is then passed to the appropriate post-link utility applications to generate the specified output file format and debugging files. All temporary files, including `l.obj`, are then deleted. The intermediate files: p-code and relocatable object files, are not deleted. This tutorial does not consider the runtime startup code that is automatically generated by the driver.

2.2.2 Generating Intermediate Files

The HI-TECH C Compiler for PIC18 MCUs version compiler uses two types of intermediate files. For C source files, the p-code file (`.p1` file) is used as the intermediate file. For assembler source files, the relocatable object file (`.obj` file) is used.

You may wish to generate intermediate files for several reasons, but the most likely will be if you are using an IDE or make system that allows an incremental build of the project. The advantage of a incremental build is that only the source files that have been modified since the last build need to be recompiled before again running the final link step. This dependency checking may result in reduced compilation times, particularly if there are a large number of source files.

You may also wish to generate intermediate files to construct your own library files, although PICC18 is capable of constructing libraries in a single step, so this is typically not necessary. See Section 2.6.49 for more information.

Intermediate files may also assist with debugging a project that fails to work as expected.

Do not use the project base name as the base name for assembly source files if you are using MPLAB IDE v8. The intermediate file produced from the C files will use the project name to form the name of the intermediate object file. This would be the same name chosen by the IDE for the intermediate object file generated for the assembly source file.

If a multi-step compilation is required the recommended compile sequence is as follows.

- Compile all modified C source files to p-code files using the `--PASS1` driver option
- Compile all modified assembler source files to relocatable object files using the `-C` driver option
- Compile all p-code and relocatable object files into a single output object file

The final step not only involves the link stage, but also code generation of all the p-code files. In effect, the code generator performs some of the tasks normally performed by the linker. Any user-specified (non standard) libraries also need to be passed to the compiler during the final step. This is the incremental build sequence used by MPLAB IDE.

TUTORIAL

MULTI-STEP COMPILATION The files in the previous example are to be compiled using a multi-step compilation. The following could be used.

```
PICC18 --chip=18F242 --pass1 main.c
PICC18 --chip=18F242 --pass1 io.c
PICC18 --chip=18F242 -c mdef.as
PICC18 --chip=18F242 main.p1 io.p1 mdef.obj sprt.obj c_sb.lpp a_sb.lib
```

If using a make system with incremental builds, only those source files that have changed since the last build need the first compilation step performed again, so not all of the first three steps need be executed.

If is important to note that the code generator needs to compile all p-code or p-code library files in the one step. Thus, if the `--PASS1` option is not used (or `--PRE` is not used), all C source files, and any p-code libraries, must be built together in the one command.

If a compilation is performed, and the source file that contains `main()` is not present in the list of C source files, an undefined symbol error for `_main` will be produced by the code generator. If the file that contains the definition for `main()` is present, but it is a subset of the C source files making up a project that is being compiled, the code generator will not be able to see the entire C program and this will defeat most of the optimization techniques employed by the code generator.

There may be multi-step compilation methods employed that lead to compiler errors as a result of the above restrictions, for example you cannot have an C function compiled into a p-code library that is called only from assembler code.

2.2.3 Special Processing

There are several special steps that take place during compilation.

2.2.3.1 Printf check

An extra execution of the code generator is performed for prior to the actual code generation phase. This pass is part of the process by which the printf library function is customized, see Section 2.3.4 for more details.

2.2.3.2 Assembly Code Requirements

After pre-processing and parsing of any C source files, but before code generation of these files, the compiler assembles any assembly source files to relocatable object files. These object files, together with any object files specified on the command line, are scanned by the compiler driver and certain information from these files are collated and passed to the code generator. Several actions are taken based on this information. See Section 3.10.4.

The driver instructs the code generator to preserve any C variables which map to symbols which are used, but not defined, in the assembly/object code. This allows variables to be defined in C code, and only every referenced in assembly code. Normally such C variables would be removed as the code generator would consider them to be used (from the C perspective). Specifically, the C variables are automatically qualified as being `volatile` which is sufficient to prevent the code generator making this optimization.

The driver also takes note of any absolute psects (viz. use the `abs` and `ovrld` `PSECT` directive flags) in the assembly/object code. The memory occupied by the psects is removed from the available memory ranges passes to the code generator and linker. This information ensures that this memory is not allocated to any C resources.

2.3 Runtime Files

In addition to the input files specified on the command line by the user, there are also compiler-generated source files and pre-compiled library files which might be compiled into the project by the driver. These are:

- Library files;
- The runtime startup module;
- The powerup routine; and
- The `printf` routine.

Strictly speaking, the powerup routine is neither compiler-generated source, nor a library routine. It is fully defined by the user, however as it is very closely associated with the runtime startup module, it is discussed with the other runtime files in the following sections.

By default, libraries appropriate for the selected driver options are automatically passed to the code generator and linker. Although individual library functions or routines will be linked in once referenced in C code, the compiler still requires the inclusion of the appropriate header file for the library function that is being used. See the appropriate library function section in Chapter A for the header file that should be used.

2.3.1 Library Files

By default, PICC18 will search the `LIB` directory of the compiler distribution for p-code library files, which are then passed to the code generator. These library files typically contain:

- The C standard library functions
- Routines implicitly called by the code generator
- Chip-specific peripherals functions
- Chip-specific memory functions

These library files are always scanned after scanning any user-specified libraries passed to the driver on the command line, thus allowing library routines to be easily replaced with user-defined alternatives. See Section 3.12.1.

The C standard libraries and libraries of implicitly-called routines can be omitted from the project by disabling the `clib` suboption of `--RUNTIME`. 2.6.55. For example:

```
--RUNTIME=default,-clib
```

If these libraries are excluded from the project then calls to any routine, or access of any variable, that is defined in the omitted library files will result in an error from the linker. The user must provide alternative libraries or source files containing definitions for any routine or symbol accessed by the project.

Do not confuse the actual library (`.lpp` or `.lib`) files and the header (`.h`) files. Both are part of a library package, but the library files contain precompiled code, typically functions and variable definitions; the header files provide declarations (as opposed to definitions) for functions, variables and types in the library files, as well as other preprocessor macros. PICC18 will always link in all the library files associated with the C standard library (unless you have used an option to prevent this), however with user-defined library packages, the inclusion of a header does not imply that the corresponding library file(s) will be searched.

2.3.1.1 Standard Libraries

The C standard libraries contain a standardised collection of functions, such as string, math and input/output routines. The range of these functions are described in Appendix A.

These libraries also contain C routines that are implicitly called by the compiler and which typically perform tasks such as floating point operations or type conversions.

The standard library name format is `stdlib-`*`compat_mode-option`*`.lpp`, where *`compat_mode`* is the compatibility mode: `htc` or `c18`; and *`option`* is any option that was used when building the library, for example `d32` for 32-bit doubles.

2.3.1.2 Utility Libraries

In addition to the C standard libraries, PICC18 automatically passes to the code generator a set of p-code libraries which contain functions that are device dependent. Such routines may, for example, access the EEPROM, or flash memory of the device.

The general form of the utility library names is *`picfamily-xp`*`.lpp`. The meaning of each field is described by:

- The family name may be the same as a specific device name or be generic, e.g. `pic18f4550` or `pic18fxx31`.
- The configuration digit, *`x`*, relates to errata information; each bit has the meaning:
  - bit #0 is true for devices implementing EEDATA errata workaround for EEPROM reads.
  - bit #1 is true for devices that implement additional NOPs when reading beyond program space at -40C.
- The library type, *`p`*, is `f` for flash libraries and `e` for eeprom libraries. The library extension is always `lpp`.

2.3.1.3 Peripheral Libraries

HI-TECH C Compiler for PIC18 MCUs has support for the MPLAB C18 peripheral library API. A native HI-TECH library is linked when the compatibility mode is the default setting (see 3.1.4), and a C18 version of this library is linked if the compiler is being run in C18 compatibility mode. These two libraries are functionally identical and conform to the same API, but differ in the source syntax.

These libraries are linked in by default. To prevent them from being searched, the `--RUNTIME=-plib` option must be used. See Section 2.6.55.

2.3.2 Runtime Startup Code

A C program requires certain objects to be initialised and the processor to be in a particular state before it can begin execution of its function `main()`. It is the job of the *runtime startup* code to perform these tasks, specifically:

- Initialisation of global variables assigned a value when defined
- Clearing of non-initialised global variables
- General setup of registers or processor state

Rather than the traditional method of linking in a generic, precompiled routine, HI-TECH C Compiler for PIC18 MCUs uses a more efficient method which actually determines what runtime startup code is required from the user's program. It does this by performing an additional link step, the output of which is used to determine the requirements of the program. From this information PICC18 then "writes" the assembler code which will perform the startup sequence.

Both the driver and code generator are involved in generating the runtime startup code. The driver takes care of device setup and this code is placed into a separate assembly startup file. The code generator handles initialization of the C environment, such as clearing uninitialized C variables and copying initialized C variables. This code is output along with the rest of the C program.

The runtime startup code is generated automatically on every compilation. If required, the assembler file which contains some of the runtime startup code can be deleted after compilation by using the driver option:

```
--RUNTIME=default,-keep
```

If the startup module is kept, it will be called `startup.as` and will be located in the current working directory. If you are using an IDE to perform the compilation the destination directory is dictated by the IDE itself, however you may use the `--OUTDIR` option to specify an explicit output directory to the compiler. The code produced by the code generator will be shown in the assembly list file associated with the project.

This is an automatic process which does not require any user interaction, however some aspects of the runtime code can be controlled, if required, using the `--RUNTIME` option. Section 2.6.55 describes the use of this option, and the following sections describes the functional aspects of the code contained in this module and its effect on program operation.

If you require any special initialization to be performed immediately after reset, you should use the *powerup* routine feature described later in Section 2.3.3.

2.3.2.1 Initialization of Data psects

One job of the runtime startup code is ensure that any initialized variables contain their initial value before the program begins execution. Initialized variables are those which are not `auto` objects and which are assigned an initial value in their definition, for example `input` in the following example.

```
int input = 88;
void main(void) { ...
```

Such initialized objects have two components: their initial value stored in a psect destined for non-volatile memory (i.e. placed in the HEX file), and space for the variable in RAM psect where the variable will reside and be accessed during program execution.

The psects used for storing these components are described in 3.8.1.

The runtime startup code will copy all the blocks of initial values from program memory to RAM so the variables will contain the correct values before main() is executed. This action can be omitted by disabling the `init` suboption of `--RUNTIME`. For example:

```
--RUNTIME=default,-init
```

With this part of the runtime startup code absent, the contents of initialized variables will be unpredictable when the program begins execution. Code relying on variables containing their initial value will fail.

Since `auto` objects are dynamically created, they require code to be positioned in the function in which they are defined to perform their initialization. It is also possible that their initial value changes on each instance of the function. As a result, initialized `auto` objects do not use the data psects and are not considered by the runtime startup code.

Variables whose contents should be preserved over a reset, or even power off, should be qualified with persistent, see Section 3.3.11.1. Such variables are linked at a different area of memory and are not altered by the runtime startup code in any way.

2.3.2.2 Clearing the Bss Psects

Those non-auto objects which are not initialized must be cleared before execution of the program begins. This task is also performed by the runtime startup code.

Uninitialized variables are those which are not auto objects and which are not assigned a value in their definition, for example output in the following example.

```
int output;
void main(void) { ...
```

Such uninitialized objects will only require space to be reserved in RAM where they will reside and be accessed during program execution (runtime).

The psects used for storing these components are described in Section 3.8.1 and typically have a name based on the initialism "bss" (Block Started by Symbol).

The runtime startup code will clear all the memory location occupied by uninitialized variables so they will contain zero before main() is executed.

Variables whose contents should be preserved over a reset should be qualified with persistent. See Section 3.3.11.1 for more information. Such variables are linked at a different area of memory and are not altered by the run- time startup code in any way.

The block clear of all the bss psects (including the memory allocated by the code generator) can be omitted by disabling the `clear` suboption of `--RUNTIME`. For example:

```
--RUNTIME=default,-clear
```

With this part of the runtime startup code absent, the contents of uninitialized variables will be unpredictable when the program begins execution.

2.3.3 The Powerup Routine

Some hardware configurations require special initialization, often within the first few instruction cycles after reset. To achieve this there is a hook to the reset vector provided via the *powerup* routine.

This routine can be supplied in a user-defined assembler module that will be executed immediately after reset. An empty powerup routine is provided in the file `powerup.as` which is located in the SOURCES directory of your compiler distribution. Refer to comments in this file for more details.

The file should be copied to your working directory, modified and included into your project as a source file. No special linker options or other code is required; the compiler will detect if you have defined a powerup routine and will automatically use it, provided the code in this routine is contained in a psect called `powerup`.

For correct operation (when using the default compiler-generated runtime startup code), the code must contain at its end a `GOTO` instruction to the label called `start`. As with all user-defined assembly code, it must take into consideration program memory paging and/or data memory banking, as well as any applicable errata issues for the device you are using. See Section 2.6.30 for more information on errata issues. The program's entry point is already defined by the runtime startup code, so this should not be specified in the powerup routine at the `END` directive (if used). See Section 4.3.10.2 for more information on this assembler directive.

2.3.4 The `printf` Routine

The code associated with the `printf` function is not found in the library files. The `printf` function is generated from a special C source file that is customized after analysis of the user's C code. See Appendix 359 for more information on the `printf` library function.

This template file is found in the LIB directory of the compiler distribution and is called `doprnt.c`. It contains a minimal implementation of the `printf` function, but with the more advanced features included as conditional code which can be utilized via preprocessor macros that are defined when it is compiled.

The parser and code generator analyze the C source code, searching for calls to the `printf` function. For all calls, the placeholders that were specified in the `printf` format strings are collated to produce a list of the desired functionality of the final function. The `doprnt.c` file is then preprocessed with the those macros specified by the preliminary analysis, thus creating a custom `printf` function for the project being compiled. After parsing, the p-code output derived from `doprnt.c` is then combined with the remainder of the C program in the final code generation step.

TUTORIAL

CALLS TO PRINTF A program contains one call to `printf`, which looks like:

```
printf("input is:  $d");
```

The compiler will note that only the `%d` placeholder is used and the doprnt module that is linked into the program will only contain code that handles printing of decimal integers. The code is latter changed and another call to `printf` is added. The new call looks like:

```
printf("output is %6d");
```

Now the compiler will detect that in addition there must be code present in the doprnt module that handles integers printed to a specific width. The code that handles this flag will be introduced into the doprnt module.

The size of the doprnt module will increase as more `printf` features are detected.

If the format string in a call to `printf` is not a string literal as in the tutorial, but is rather a pointer to a string, then the compiler will not be able to reliably predict the `printf` usage, and so it forces a more complete version of `printf` to be generated. However, even without being able to scan `printf` placeholders, the compiler can still make certain assumptions regarding the usage of the function. In particular, the compiler can look at the number and type of the additional arguments to `printf` (those following the format string expression) to determine which placeholders could be valid. This enables the size and complexity of the generated `printf` routine to be kept to a minimum.

TUTORIAL

PRINTF WITHOUT LITERAL FORMAT STRINGS If there is only one reference to `printf` in a program and it appears as in the following code:

```
void my_print(const char * mes) {
  printf(mes);
}
```

the compiler cannot determine the exact format string, but can see that there are no additional arguments to `printf` following the format string represented by `mes`. Thus, the only valid format strings will not contain placeholders that print any arguments, and a minimal version of `printf` will be generated and compiled. If the above code was rewritten as:

```
void my_print(const char * mes, double val) {
  printf(mes, val);
}
```

the compiler will detect that the argument being printed has `double` type, thus the only valid placeholders would be those that print floating point types, for example `%e`, `%f` and `%g`.

No aspect of this operation is user-controllable (other than by adjusting the calls to `printf`), however the actual `printf` code used by a program can be observed. If compiling a program using `printf`, the driver will leave behind the pre-processed version of `doprnt.c`. This module, called `doprnt.pre` in your working directory, will show the C code that will actually be contained in the `printf` routine. As this code has been pre-processed, indentation and comments will have been stripped out as part of the normal actions taken by the C pre-processor.

2.4 Debugging Information

Several driver options and output files allow development tools, such as HI-TIDE™ or MPLAB®, to perform source-level debugging of the output code.

The default behaviour of the PICC18 command is to produce a *Microchip* COFF and *Intel* HEX output. If no output filename or type is specified, PICC18 will produce these files with the same base name as the first source or object file specified on the command line. Table 2.12 shows the output format options available with PICC18. The *File Type* column lists the filename extension which will be used for the output file.

In addition to the options shown, the `-O` option may be used to request generation of binary or UBROF files. If you use the `-O` option to specify an output filename with a `.bin` type, for example `-Otest.bin`, PICC18 will produce a binary file. Likewise, if you need to produce UBROF files, you can use the -O option to specify an output file with type `.ubr`, for example `-Otest.ubr`.

2.5 Compiler Messages

All compiler applications, including the command-line driver, PICC18, use textual messages to report feedback during the compilation process. A centralized messaging system is used to produce the messages which allows a consistancy during all stages of the compilation process.

2.5.1 Messaging Overview

A message is referenced by a unique number which is passed to the alert system by the compiler application that needs to convey the information. The message string corresponding to this number is obtained from Message Description Files (MDF) which are stored in the `DAT` directory of the compiler distribution.

When a message is requested by a compiler application, its number is looked up in the MDF which corresponds to the currently selected language. The language of messages can be altered as discussed in Section 2.5.2.

Once found, the alert system determines the message type that should be used to display the message. There are several different message types which are described in Section 2.5.3. The default type is stored in the MDF, however this can be overridden by the user, as described in Section 2.5.3. The user is also able to set a threshold for warning message importance, so that only those which the user considers significant will be displayed. In addition, messages with a particular number can be disabled. Both of these methods are explained in Section 2.5.5.1.

Provided the message is enabled and it is not a warning messages that is below the warning threshold, the message string will be displayed.

In addition to the actual message string, there are several other pieces of information that may be displayed, such as the message number, the name of the file for which the message is applicable, the file's line number and the application that requested the message, etc.

If a message is being displayed as an error, a counter is incremented. After a certain number of errors has been reached, compilation of the current module will cease. The default number of errors that will cause this termination can be adjusted by using the `--ERRORS` option, see Section 2.6.32. This counter is reset after each compilation step of each module, thus specifying a maximum of five errors will allow up to five errors from the parser, five from the code generator, five from the linker, five from the driver, etc.

If a language other than English is selected, and the message cannot be found in the appropriate non-English MDF, the alert system tries to find the message in the English MDF. If an English message string is not present, a message similar to:

```
error/warning (*) generated, but no description available
```

where * indicates the message number that was generated, will be printed, otherwise the message in the requested language will be displayed.

Table 2.2: Support languages

Language	MDF name
English	en_msgs.txt
German	de_msgs.txt
French	fr_msgs.txt

2.5.2 Message Language

HI-TECH C Compiler for PIC18 MCUs Supports more than one language for displayed messages. There is one MDF for each language supported.

The language used for messaging may be specified with each compile using the --LANG option, see Section 2.6.39. Alternatively it may be set up in a more permanent manner by using the --LANG option together with the --SETUP option which will store the default language in either the registry, under Windows, or in a configuration file on other systems. On subsequent builds the default language used will be that specified.

Table shows the MDF applicable for the currently supported languages.

2.5.3 Message Type

There are four types of message whose default behaviour is described below.

Advisory Messages convey information regarding a situation the compiler has encountered or some action the compiler is about to take. The information is being displayed "for your interest" and typically require no action to be taken.
Unless prevented by some driver option or another error message, the project will be linked and the requested output file(s) will be generated.

Warning Messages indicate source code or some other situation that is valid, but which may lead to runtime failure of the code. The code or situation that triggered the warning should be investigated, however, compilation of the current module will continue, as will compilation of any remaining modules.
Unless prevented by some driver option or another error message, the project will be linked and the requested output file(s) will be generated.

Error Messages indicate source code that is illegal and that compilation of this code either cannot or will not take place. Compilation will be attempted for the remaining source code in the current module, but no additional modules will be compiled and the compilation process will then conclude.
The requested output files will not be produced.

Table 2.3: Messaging environment variables

Variable	Effect
HTC_MSG_FORMAT	All advisory messages
HTC_WARN_FORMAT	All warning messages
HTC_ERR_FORMAT	All error and fatal error messages

Fatal Error Messages indicate a situation that cannot allow compilation to proceed and which required the the compilation process to stop immediately.
The requested output files will not be produced.

2.5.4 Message Format

By default, messages are printed in the most useful human-readable format as possible. This format can vary from one compiler application to another, since each application reports information about different file formats. Some applications, for example the parser, are typically able to pinpoint the area of interest down to a position on a particular line of C source code, whereas other applications, such as the linker, can at best only indicate a module name and record number, which is less directly associated with any particular line of code. Some messages relate to driver options which are in no way associated with any source code.

There are several ways of changing the format in which message are displayed, which are discussed below.

The driver option -E (with or without a filename) alters the format of all displayed messages. See Section 2.6.4. Using this option produces messages that are better suited to machine parsing, and user-friendly. Typically each message is displayed on a single line. The general form of messages produced with the -E option in force is:

```
filename line_number: (message number) message string (message type)
```

The -E option also has another effect. If it is being used, the driver first checks to see if special environment variables have been set. If so, the format dictated by these variables are used as a template for all messages produced by all compiler applications. The names of these variables are given in Table 2.3.

The value of these environment variables are strings that are used as templates for the message format. Printf-like placeholders can be placed within the string to allow the message format to be customised. The placeholders and what they represent are indicated in Table 2.4.

If these options are used in a DOS batch file, two percent characters will need to be used to specify the placeholders, as DOS interprets a single percent character as an argument and will not pass this on to the compiler. For example:

Table 2.4: Messaging placeholders

Placeholder	Replacement
%a	application name
%c	column number
%f	filename
%l	line number
%n	message number
%s	message string (from MDF)

```
--ERRFORMAT="file %%f:  line %%l"
```

The message environment variables, in turn, may be overridden by the driver options: --MSGFORMAT, --WARNFORMAT and --ERRFORMAT, see Sections 2.6.31, 2.6.43 and 2.6.65. These options take a string as their argument. The option strings are formatted, and can use the same placeholders, as their variable counterparts.

TUTORIAL

CHANGING MESSAGE FORMATS A project is compiled, but produces a warning from the parser and an error from the linker. By default the following messages are displayed when compiling.

```
main.c:  main()
17:  ip = &b;
^ (362) redundant "&" applied to array (warning)
(492) attempt to position absolute psect "text" is illegal
```

Notice that the format of the messages from the parser and linker differ since the parser is able to identify the particular line of offending source code. The parser has indicated the name of the file, indicated the function in which the warning is located, reproduced the line of source code and highlighted the position at which the warning was first detected, as well as show the actual warning message string.

The -E option is now used and the compiler issues the same messages, but in a new format as dictated by the -E option. Now environment variables are set and no other messaging driver options were specified so the default -E format is used.

```
main.c:  12:  (362) redundant "&" applied to array (warning)
(492) attempt to position absolute psect "text" is illegal (error)
```

Notice that now all message follow a more uniform format and are displayed on a single line.

The user now sets the environment variable `HTC_WARN_FORMAT` to be the following string. (Under Windows, this can be performed via the Control Panel's System panel.)

```
%a %n %l %f %s
```

and the project recompiled. The following output will be displayed.

```
parser 362 12 main.c redundant "&" applied to array (492)
attempt to position absolute psect "text" is illegal (error)
```

Notice that the format of the warning was changed, but that of the error message was not. The warning format now follows the specification of the environment variable. The application name (parser) was substituted for the `%a` placeholder, the message number (362) substituted the `%n` placeholder, etc.

The option `--ERRFORMAT="%a %n %l %f %s"` is then added to the driver command line and the following output is observed.

```
parser 362 12 main.c redundant "&" applied to array
linker 492   attempt to position absolute psect "text" is illegal
```

Note that now the warning and error formats have changed to that requested. For the case of the linker error, there is no line number information so the replacement for this placeholder is left blank.

2.5.5 Changing Message Behaviour

Both the attributes of individual messages and general settings for messaging system can be modified during compilation. There are both driver command-line options and C pragmas that can be used to achieve this.

2.5.5.1 Disabling Messages

Each warning message has a default number indicating a level of importance. This number is specified in the MDF and ranges from -9 to 9. The higher the number, the more important the warning.

Warning messages can be disabled by adjusting the warning level threshold using the `--WARN` driver option, see Section 2.6.64. Any warnings whose level is below that of the current threshold are not displayed. The default threshold is 0 which implies that only warnings with a warning level of 0 or higher will be displayed by default. The information in this option is propagated to all compiler applications, so its effect will be observed during all stages of the compilation process.

Warnings may also be disabled by using the `--MSGDISABLE` option, see Section 2.6.42. This option takes a comma-separated list of message numbers. Any warnings which are listed are disabled and will never be issued, regardless of any warning level threshold in place. This option cannot be used to disable error messages.

Some warning messages can also be disabled by using the `warning disable` pragma. This pragma will only affect warnings that are produced by either parser or the code generator, i.e. errors directly associated with C code. See Section 3.11.4.6 for more information on this pragma.

Error messages can also be disabled, however a slighty more verbose form of the command is required to confirm the action required. To specify an error message number in the `--MSGDISABLE` command, the number must be followed by `:off` to ensure that it is actually disabled. For example: `--MSGDISABLE=195:off` will disable error number 195.

Disabling error or warning messages in no way fixes any potential problems reported by the message. Always use caution when exercising this option.

2.5.5.2 Changing Message Types

It is also possible to change the type of some messages. This is only possible by the use of the `warning pragma` and only affects messages generated by the parser or code generator. See Section 3.11.4.6 for more information on this pragma.

2.6 PICC18 Driver Option Descriptions

Most aspects of the compilation can be controlled using the command-line driver, PICC18. The driver will configure and execute all required applications, such as the code generator, assembler and linker.

PICC18 recognizes the compiler options listed in the table below and which are described in the sections that follow. The case of the options is not important, however command shells in UNIX-based operating systems are case sensitive when it comes to names of files.

2.6.1 Option Formats

All single letter options are identified by a leading *dash* character, "–", e.g. `-C`. Some single letter options specify an additional data field which follows the option name immediately and without any whitespace, e.g. `-Ddebug`.

Multi-letter, or word, options have two leading *dash* characters, e.g. `--ASMLIST`. (Because of the double *dash*, you can determine that the option `--ASMLIST`, for example, is not a `-A` option followed by the argument `SMLIST`.)

Some of these options define suboptions which typically appear as a *comma*-separated list following an *equal* character, `=`, e.g. `--OUTPUT=hex,cof`. The exact format of the options varies and are described in detail in the following sections.

Some commonly used suboptions include `default`, which represent the default specification that would be used if this option was absent altogether; `all`, which indicates that all the available suboptions should be enabled as if they had each been listed; and `none`, which indicates that all suboptions should be disabled. Some suboptions may be prefixed with a plus character, `+`, to indicate that they are in addition to the other suboptions present, or a minus character "`-`", to indicate that they should be excluded. In the following sections, *angle brackets,* < >, are used to indicate optional parts of the command.

See the –HELP option, Section 2.6.36, for more information about options and suboptions.

2.6.2 `-C`: Compile to Object File

The `-C` option is used to halt compilation after generating a relocatable object file. This option is frequently used when compiling assembly source files using a "make" utility. Use of this option when only a subset of all the C source files in a project are being compiled will result in an error from the code generator. See Section 2.2.2 for more information on generating and using intermediate files.

2.6.3 `-Dmacro`: Define Macro

The `-D` option is used to define a preprocessor macro on the command line, exactly as if it had been defined using a `#define` directive in the source code. This option may take one of two forms, `-Dmacro` which is equivalent to:

```
#define  macro  1
```

placed at the top of each module compiled using this option, or `-Dmacro=`*`text`* which is equivalent to:

```
#define  macro  text
```

where *`text`* is the textual substitution required. Thus, the command:

```
PICC18 --CHIP=18F242 -Ddebug -Dbuffers=10 test.c
```

will compile `test.c` with macros defined exactly as if the C source code had included the directives:

```
#define  debug    1
#define  buffers 10
```

See Section 2.7 for use of this option in MPLAB IDE.

2.6.4 -E*file*: Redirect Compiler Errors to a File

This option has two purposes. The first is to change the format of displayed messages. The second is to optionally allow messages to be directed to a file as some editors do not allow the standard command line redirection facilities to be used when invoking the compiler.

The general form of messages produced with the -E option in force is:

```
filename  line_number:  (message number)  message string  (message type)
```

If a filename is specified immediately after -E, it is treated as the name of a file to which all messages (errors, warnings etc) will be printed. For example, to compile x.c and redirect all errors to x.err, use the command:

```
PICC18 --CHIP=18F242 -Ex.err x.c
```

The -E option also allows errors to be appended to an existing file by specifying an *addition* character, +, at the start of the error filename, for example:

```
PICC18 --CHIP=18F242 -E+x.err y.c
```

If you wish to compile several files and combine all of the errors generated into a single text file, use the -E option to create the file then use -E+ when compiling all the other source files. For example, to compile a number of files with all errors combined into a file called project.err, you could use the -E option as follows:

```
PICC18 --CHIP=18F242 -Eproject.err -O --PASS1 main.c
PICC18 --CHIP=18F242 -E+project.err -O --PASS1 part1.c
PICC18 --CHIP=18F242 -E+project.err -C asmcode.as
```

Section 2.5 has more information regarding this option as well as an overview of the messaging system and other related driver options.

2.6.5 -G*file*: Generate Source-level Symbol File

The -G option generates a *source-level symbol file* (i.e. a file which allows tools to determine which line of source code is associated with machine code instructions, and determine which source-level variable names correspond with areas of memory, etc.) for use with supported debuggers and simulators such as MPLAB IDE. If no filename is given, the symbol file will have the same base name as

the project name (see Section 2.1), and an extension of `.sym`. For example the option `-Gtest.sym` generates a symbol file called `test.sym`. Symbol files generated using the `-G` option include source-level information for use with source-level debuggers.

Note that all source files for which source-level debugging is required should be compiled with the `-G` option. The option is also required at the link stage, if this is performed separately. For example:

```
PICC18 --CHIP=18F242 -G --PASS1 test.c modules1.c
PICC18 --CHIP=18F242 -Gtest.sym test.p1 module1.p1
```

The `--IDE` option, see Section 2.6.38 will typically enable the `-G` option.

2.6.6 `-I`*`path`*: Include Search Path

Use `-I` to specify an additional directory to use when searching for header files which have been included using the `#include` directive. The `-I` option can be used more than once if multiple directories are to be searched.

The default include directory containing all standard header files is always searched even if no `-I` option is present. The default search path is searched after any user-specified directories have been searched. For example:

```
PICC18 --CHIP=18F242 -C -Ic:\include -Id:\myapp\include test.c
```

will search the directories `c:\include` and `d:\myapp\include` for any header files included into the source code, then search the default include directory (the include directory where the compiler was installed).

It is strongly advised not to use `-I` to add the compiler's default include path, not only because it is unnecessary but in the event that the build tool changes, the path specified here will be searched prior to searching the new compiler's default path.

This option has no effect for files that are included into assembly source using the `INCLUDE` directive. See Section 4.3.11.4.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.7 `-L`*`library`*: Scan Library

The `-L` option is used to specify additional libraries which are to be scanned by the linker and code generator. Libraries specified using the `-L` option are scanned before any C standard libraries.

The argument to `-L` is a library keyword to which the prefix `pic8` and other letters and digits, as described in Section 2.3.1, are added. Both a p-code and object code library filename is generated

and passed to the code generator and linker, respectively. The case of the string following the option is important for environments where filenames are case sensitive.

Thus the option `-Lt` when compiling for a 18F452 will, for example, specify the library filenames `pic86l-t.lpp` and `pic86l-t.lib`. The option `-Lxx` will specify libraries called `pic86l-xx.lpp` and `pic86l-xx.lib`. All libraries must be located in the `LIB` subdirectory of the compiler installation directory.

If you wish the linker to scan libraries whose names do not follow the above naming convention or whose locations are not in the `LIB` subdirectory, simply include the libraries' names on the command line along with your source files.

The commonly-used PICC-18 Standard compiler options `-Ll`, `-Lf` and `-Lw` should not be used for altering the behaviour of the `printf` function. The library files corresponding to these opptions are not provided with the PRO version of this compiler, and an error will result if these options are used with creating these library sets. A custom `printf` function is automatically generated by the compiler when required, as described in section 2.3.4.

2.6.8 `-L-option`: Adjust Linker Options Directly

The `-L` driver option can also be used to specify an option which will be passed directly to the linker. If `-L` is followed immediately by text starting with a *dash* character "-", the text will be passed directly to the linker without being interpreted by PICC18. For example, if the option `-L-FOO` is specified, the `-FOO` option will be passed on to the linker. The linker will then process this option, when, and if, it is invoked, and perform the appropriate function, or issue an error if the option is invalid.

Take care with command-line options. The linker cannot interpret driver options; similarly the command-line driver cannot interpret linker options. In most situations, it is always the command-line driver, PICC18, that is being executed. If you need to add alternate settings in the linker tab in an MPLAB `Build options...` dialogue, these are the *driver* options (not linker options), but which are used by the driver to generate the appropriate linker options during the linking process.

The -L option is especially useful when linking code which contains non-standard program sections (or psects), as may be the case if the program contains assembly code which contains user-defined psects. Without this -L option, it would be necessary to invoke the linker manually to allow the linker options to be adjusted.

One commonly used linker option is -N, which sorts the symbol table in the map file by address, rather than by name. This would be passed to PICC18 as the option -L-N.

This option can also be used to replace default linker options: If the string starting from the first character after the -L up to the first = character matches first part of a default linker option, then that default linker option is replaced by the option specified by the -L.

TUTORIAL

REPLACING DEFAULT LINKER OPTIONS In a particular project, the psect entry is used, but the programmer needs to ensure that this psect is positioned above the address 800h. This can be achieved by adjusting the default linker option that positions this psect. First, a map file is generated to determine how this psect is normally allocated memory. The Linker command line: in the map file indicates that this psect is normally linked using the linker option:

```
-pentry=CODE
```

Which places entry anywhere in the memory defined by the CODE class. The programmer then re-links the project, but now using the driver option:

```
-L-pentry=CODE+800h
```

to ensure that the psect is placed above 800h. Another map file is generated and the Linker command line: section is checked to ensure that the option was recieved and executed by the linker. Next, the address of the psect entry is noted in the psect lists that appear later in the map file. See Section 5.9 for more information on the contents of the map file.

If there are no characters following the first = character in the -L option, then any matching default linker option will be deleted. For example: -L-pfirst= will remove any default linker option that begins with the string -pfirst=. No warning is generated if such a default linker option cannot be found.

TUTORIAL

ADDING AND DELETING DEFAULT LINKER OPTIONS The default linker options for for a project links several psects in the following fashion.

```
-pone=600h,two,three
```

which links one at 600h, then follows this with two, then three. It has been decided that the psects should be linked so that one follows two, which follows three, and

that the highest address of `one` should be located at 5FFh. This new arragement can be specified issuing the following driver option:
`-L-pthree=-600h,two,one`
which creates passes the required linker options to the linker. The existing default option is still present, so this must be removed by use the driver option:
`-L-pone=`
which will remove the existing option.

The default option that you are deleting or replacing must contain an *equal* character.

2.6.9 `-Mfile`: Generate Map File

The `-M` option is used to request the generation of a map file. The map is generated by the linker an includes detailed information about where objects are located in memory, see Section 5.9 for information regarding the content of map files.

If no filename is specified with the option, then the name of the map file will have the project name, with the extension `.map`.

2.6.10 `-Nsize`: Identifier Length

This option is currently not implemented. The identifier size is fixed at 255, but can be changed to 31 by using the `--STRICT` option, see 2.6.60.

2.6.11 `-Ofile`: Specify Output File

This option allows the basename of the output file(s) to be specified. If no `-O` option is given, the output file(s) will be named after the first source or object file on the command line. The files controlled are any produced by the linker or applications run subsequent to that, e.g. `CROMWELL`. So for instance the HEX file, MAP file and SYM file are all controlled by the `-O` option.

The `-O` option can also change the directory in which the output file is located by including the required path before the filename, e.g. `-Oc:\project\output\first`. This will then also specify the output directory for any files produced by the linker or subsequently run applications. Any relative paths specified are with respect to the current working directory.

Any extension supplied with the filename will be ignored. The name and path specified by the `-O` option will apply to all output files.

The options that specify MAP file creation (`-M`, see 2.6.9), and SYM file creation (`-G`, see 2.6.5) override any name or path information provided by `-O` relevant to the MAP and SYM file.

To change the directory in which all output and intermediate files are written, use the `--OUTDIR` option, see Section 2.6.48. Note that if `-O` specifies a path which is inconsistent with the path specified in the `--OUTDIR` option, this will result in an error.

2.6.12 `-P`: Preprocess Assembly Files

The `-P` option causes the assembler files to be preprocessed before they are assembled thus allowing the use of preprocessor directives, such as `#include`, with assembler code. By default, assembler files are not preprocessed.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.13 `-Q`: Quiet Mode

This option places the compiler in a *quiet mode* which suppresses the HI-TECH Software copyright notice from being displayed.

2.6.14 `-S`: Compile to Assembler Code

The `-S` option stops compilation after generating an assembler source file. An assembler file will be generated for each C source file passed on the command line. The command:

```
PICC18 --CHIP=18F242 -S test.c
```

will produce an assembler file called `test.as` which contains the code generated from `test.c`. This option is particularly useful for checking function calling conventions and signature values when attempting to write external assembly language routines.

The file produced by this option differs to that produced by the `--ASMLIST` option in that it does not contain op-codes or addresses and it may be used as a source file and subsequently passed to the assembler to be assembled.

2.6.15 `-Umacro`: Undefine a Macro

The `-U` option, the inverse of the `-D` option, is used to *undefine* predefined macros. This option takes the form `-Umacro`. The option, `-Udraft`, for example, is equivalent to:

```
#undef  draft
```

placed at the top of each module compiled using this option.

See Section 2.7 for use of this option in MPLAB IDE.

Table 2.5: Compiler Responses to Memory Qualifiers

Selection	Response
require	The qualifiers will be honored. If they cannot be met, an error will be issued.
request	The qualifiers will be honored, if possible. No error will be generated if they cannot be followed
ignore	The qualifiers will be ignored and code compiled as if they were not used.
reject	If the qualifiers are encountered, an error will be immediately generated.

2.6.16 -V: Verbose Compile

The -V is the *verbose* option. The compiler will display the command lines used to invoke each of the compiler applications or compiler passes. Displayed will be the name of the compiler application being executed, plus all the command-line arguments to this application. This option may be useful for determining the exact linker options if you need to directly invoke the HLINK command.

If this option is used twice, it will display the full path to each compiler application as well as the full command line arguments. This would be useful to ensure that the correct compiler installation is being executed if there is more than one installed.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.17 -X: Strip Local Symbols

The option -X strips local symbols from any files compiled, assembled or linked. Only global symbols will remain in any object files or symbol files produced.

2.6.18 --ADDRQUAL: Set Compiler Response to Memory Qualifier

The --ADDRQUAL option indicates the compiler's response to some of the non-standard memory qualifiers in C source code.

By default these qualifiers are ignored, i.e. they are accepted without error, but have no effect. Using this option allows these qualifiers to be interpreted differently by the compiler.

The qualifiers affected by this option are the near and far. The bank*x* qualifiers (bank0, bank1, bank2 etc.), are not currently affected by this option.

The suboptions are detailed in Table 2.5.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.19 --ASMLIST: Generate Assembler .LST Files

The --ASMLIST option tells PICC18 to generate one or more *assembler listing file* for each C and assembly source module being compiled.

In the case of code being assembled that was originally C source, the list file shows both the original C code and the corresponding assembly code generated by the code generator. For both C and assembly source code, a line number, the binary op-codes and addresses are shown. If the assembler optimizer is enabled (default operation) the list file may differ from the original assembly source code. The assembler optimizer may also simpify some expression and remove some assembler directives from the listing file for clarity, although they are processed in the usual way.

Provided the link stage has successfully concluded, the listing file will be updated by the linker so that it contains absolute addresses and symbol values. Thus you may use the assembler listing file to determine the position of, and exact op codes corresponding to, instructions.

2.6.20 `--CHECKSUM=`*`start-end@destination<,specs>`*: Calculate a checksum

This option will perform a checksum over the address range specified and store the result at the destination address specified. Additional specifications can be appended as a comma separated list to this option. Such specifications are:

,width=*n* select the byte-width of the checksum result. A negative width will store the result in little-endian byte order. Result widths from one to four bytes are permitted.

,offset=*nnnn* An initial value or offset to be added to this checksum.

,algorithm=*n* Select one of the checksum algorithms implemented in hexmate. The selectable algorithms are described in Table 5.10.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.21 `--CHIP=`*`processor`*: Define Processor

This option is the only option that is mandatory. It specifies the target processor for the compilation.

To see a list of supported processors that can be used with this option, use the `--CHIPINFO` option described in Section 2.6.22.

See also Section 4.3.10.20 for information on setting the target processor from within assembly files.

2.6.22 `--CHIPINFO`: Display List of Supported Processors

The `--CHIPINFO` option simply displays a list of processors the compiler supports. The names listed are those chips defined in the chipinfo file and which may be used with the `--CHIP` option.

Table 2.6: Compatibility modes

Mode	Operation
htc	HI-TECH C (default)
c18	MPLAB C18 traditional
c18e	MPLAB C18 Extended

2.6.23 --CMODE: Specify compatibility mode

This option allows the compiler to be run in a special compatibility mode. The modes are given in Table 2.6. This option will be automatically specified with HI-TECH C Compiler for PIC18 MCUs when the mcc18 compatibility driver is employed. It is not recommended that this option be used explicitly. See 3.1.4 for more information on building legacy projects.

2.6.24 --CODEOFFSET: Offset Program Code to Address

In some circumstances, such as bootloaders, it is necessary to shift the program image to an alternative address. This option is used to specify a base address for the program code image. With this option, all code psects (including interrupt vectors and constant data) that the linker would ordinarily control the location of, will be adjusted.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.25 --CR=*file*: Generate Cross Reference Listing

The --CR option will produce a *cross reference listing*. If the *file* argument is omitted, the "raw" cross reference information will be left in a temporary file, leaving the user to run the CREF utility. If a filename is supplied, for example --CR=test.crf, PICC18 will invoke CREF to process the cross reference information into the listing file, in this case test.crf. If multiple source files are to be included in the cross reference listing, all must be compiled and linked with the one PICC18 command. For example, to generate a cross reference listing which includes the source modules main.c, module1.c and nvram.c, compile and link using the command:

```
PICC18 --CHIP=18F242 --CR=main.crf main.c module1.c nvram.c
```

Thus this option can not be used when using any compilation process that compiles each source file separately using the -C or --PASS1 options. Such is the case for most IDEs, including MPLAB IDE, and makefiles.

Table 2.7: Supported Double Types

Suboption	Type
24	Truncated IEEE754 24-bit doubles
32	IEEE754 32-bit doubles

2.6.26 --DEBUGGER=*type*: Select Debugger Type

This option is intended for use for compatibility with debuggers. PICC18supports the Microchip ICD2 debugger and using this option will configure the compiler to conform to the requirements of the ICD2 (reserving memory addresses, etc.). For example:

```
PICC18 --CHIP=18F242 --DEBUGGER=icd2 main.c
```

Basic debugging with Microchip REALICE, ICD3, PICKIT2 and PICKIT3 are also supported when `--debugger=realice`, `--debugger=icd3`, `--debugger=pickit2` or `--debugger=pickit3` is used.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.27 --DOUBLE=*type*: Select kind of Double Types

This option allows the kind of double types to be selected. By default the compiler will choose the truncated IEEE754 24-bit implementation for double types. With this option, this can be changed to 32-bits.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.28 --ECHO: Echo command line before processing

Use of this option will result in the command line being echoed to the `stderr` stream before compilation is commenced. Each token of the command line will be printed on a separate line and will appear in the order in which they are placed on the command line.

2.6.29 --EMI=*type*: Select operating mode of the external memory interface (EMI)

Those PIC18 devices which can interface with an external memory are capable of operating in several modes. The mode selected is determined by the type of memory available and the connection method used. The interface can operate in 16-bit modes; *word write* and *byte select* mode or in an 8-bit mode: *byte write* mode. Valid types that can be specified to this option are: `wordwrite`,

byteselect or bytewrite. Which mode is selected will affect the code generated when writing to the external data. In word write mode, dummy reads and writes may be added to ensure that an even number of bytes are always written. In byte select or byte write modes dummy reads and writes are not generated and can result in more efficient code. Note that this option does not in any way pre-configure the device for operation in the selected mode.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.30 --ERRATA=*type*: Specify to add or remove specific errata workarounds

This option allows specification of the types of software workarounds to apply in order to overcome documented silicon errata issues. The chip configuration file nominates a default set of errata issues that apply to each device. To compile for an ideal chip, that is, apply no additional workarounds use --ERRATA=none.

2.6.31 --ERRFORMAT=*format*: Define Format for Compiler Messages

If the --ERRFORMAT option is not used, the default behaviour of the compiler is to display any errors in a "human readable" format line. This standard format is perfectly acceptable to a person reading the error output, but is not generally usable with environments which support compiler error handling. The following sections indicate how this option may be used in such situations.

This option allows the exact format of printed error messages to be specified using special place-holders embedded within a message template. See Section 2.5 for full details of the messaging system employed by PICC18.

This section is also applicable to the --WARNFORMAT and --MSGFORMAT options which adjust the format of warning and advisory messages, respectively.

See Section 2.6.39 for the appropriate option to change the message language.

2.6.32 --ERRORS=*number*: Maximum Number of Errors

This option sets the maximum number of errors each compiler application, as well as the driver, will display before stopping. By default, up to 20 error messages will be displayed. See Section 2.5 for full details of the messaging system employed by PICC18.

2.6.33 --FILL=*opcode*: Fill Unused Program Memory

This option allows specification of a hexadecimal opcode that can be used to fill all unused program memory locations with a known code sequence. See the HEXMATE Section for details of the option format and features. Multi-byte codes should be entered in little endian byte order.

See Section 2.7 for use of this option in MPLAB IDE.

Table 2.8: Supported Float Types

Suboption	Type
double	Size of float matches size of double type
24	Truncated IEEE754 24-bit float
32	IEEE754 32-bit float

2.6.34 --FLOAT=*type*: Select kind of Float Types

This option allows the kind of float types to be selected. By default the compiler will choose the truncated IEEE754 24-bit implementation for float types. With this option, this can be changed to 32-bits.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.35 --GETOPTION=*app*,*file*: Get Command-line Options

This option is used to retrieve the command line options which are used for named compiler application. The options are then saved into the given file. This option is not required for most projects.

2.6.36 --HELP<=*option*>: Display Help

The --HELP option displays information on the PICC18 compiler options. To find out more about a particular option, use the option's name as a parameter. For example:

```
PICC18 --help=warn
```

This will display more detailed information about the --WARN option, the available suboptions, and which suboptions are enabled by default.

2.6.37 --HTML: Generate HTML Debug Files

This option will generate a series of HTML files that can be used to explore the compilation results of the current project. The files are located in a directory called html, placed in the output directory. The top level file (that can be opened with your favourite web browser) is called index.html. Use this option at all stages of compilation.

The index page is a graphical representation of the compilation process. Each file icon is clickable and will open with the contents of that file (even intermediate files, and binary files open in a human-readable form), and each application icon can also be clicked to show a page containing information about that application's invocation and results.

Table 2.9: Supported IDEs

Suboption	IDE
`hitide`	HI-TECH Software's HI-TIDE
`mplab`	Microchip's MPLAB

Table 2.10: Supported languages

Suboption	Language
`en, english`	English
`fr, french,francais`	French
`de, german, deutsch`	German

The list of all preprocessor macros (preprocessor icon) and the graphical memory usage map (Linker icon) provide information that is not otherwise readily accessible.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.38 `--IDE=type`: Specify the IDE being used

This option is used to automatically configure the compiler for use by the named Integrated Development Environment (IDE). The supported IDE's are shown in Table 2.9.

2.6.39 `--LANG=language`: Specify the Language for Messages

This option allows the compiler to be configured to produce error, warning and some advisory messages in languages other than English. English is the default language and some messages are only ever printed in English regardless of the language specified with this option.

Table 2.10 shows those langauges currently supported.

See Section 2.5 for full details of the messaging system employed by PICC18.

2.6.40 `--MEMMAP=file`: Display Memory Map

This option will display a memory map for the specified map file. This option is seldom required, but would be useful if the linker is being driven explicitly, i.e. instead of in the normal way through the driver. This command would display the memory summary which is normally produced at the end of compilation by the driver.

2.6.41 --MODE=*mode*: Choose Compiler Operating Mode

This option selects the basic operating mode of the compiler. The available types are `pro` and `lite`. A compiler operating in PRO mode uses full optimization and produces the smallest code size. Standard mode uses limited optimizations, and LITE mode only uses a minimum optimization level and will produce relatively large code.

Only those modes permitted by the compiler license status will be accepted. For example if you have purchased a Standard compiler license, that compiler may be run in Standard or Lite mode, but not the PRO mode.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.42 --MSGDISABLE=*messagelist*: Disable Warning Messages

This option allows warning or advisory messages to be disabled during compilation of all modules within the project, and during all stages of compilation. Warning mesasges can also be disabled using pragma directives. For full information on the compiler's messaging system, see Section 2.5.

The `messagelist` is a comma-separated list of warning numbers that are to be disabled. If the number of an error is specified, it will be ignored by this option. If the message list is specified as `0`, then all warnings are disabled.

2.6.43 --MSGFORMAT=*format*: Set Advisory Message Format

This option sets the format of advisory messages produced by the compiler. See Section 2.5 for full information.

2.6.44 --NODEL: Do not Remove Temporary Files

Specifying `--NODEL` when building will instruct PICC18not to remove the intermediate and temporary files that were created during the build process.

2.6.45 --NOEXEC: Don't Execute Compiler

The `--NOEXEC` option causes the compiler to go through all the compilation steps, but without actually performing any compilation or producing any output. This may be useful when used in conjunction with the `-V` (verbose) option in order to see all of the command lines the compiler uses to drive the compiler applications.

Table 2.11: Optimization Options

Option name	Function
1..9	Select global optimization level (1 through 9)
asm	Select optimizations of assembly derived from C source
asmfile	Select optimizations of assembly source files
debug	Favour accurate debugging over optimization
space	Favour optimization of code for space over speed (default) (PRO mode only)
speed	Favour optimization of code for speed over space (PRO mode only)
all	Enable all compiler optimizations (also includes space)
none	Do not use any compiler optimziations

2.6.46 --OBJDIR=*dir*: Specify a Directory for Intermediate Files

This option allows a directory to be nominated in for PICC18to locate its intermediate files. Intermediate file include .pre and .p1 file for C source, and also includes .obj and .lst files for assembly source and the compiler-generated runtime startup source file.

If this option is omitted, intermediate files will be created in the current working directory. This option will not set the location of output files, instead use --OUTDIR. See 2.6.48 and 2.6.11 for more information.

2.6.47 --OPT<=*type*>: Invoke Compiler Optimizations

The --OPT option allows control of all the compiler optimizers. By default, without this option, all optimizations are enabled. The options --OPT or --OPT=all also enable all optimizations. Optimizations may be disabled by using --OPT=none, or individual optimizers may be controlled, e.g. --OPT=asm will only enable some assembler optimizations. Table 2.11 lists the available optimization types. The optimizations that are controlled through specifying a level 1 through 9 affect register-allocation optimization during the code generation stage. The level selected is commonly referred to as the *global optimization level*; however this has virtally no effect on compilation for PIC18 devices.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.48 --OUTDIR=*path*: Specify a Directory for Output Files

This option allows a directory to be nominated in for PICC18to locate its output files. If this option is omitted, output files will be created in the current working directory. This option will not set the location of intermediate files, instead use --OBJDIR. See 2.6.46 and 2.6.11 for more information.

Table 2.12: Output file formats

Type tag	File format
`lib`	Library File
`lpp`	P-code library
`intel`	*Intel* HEX
`inhx032`	Intel HEX with upper address initialization of zero
`tek`	Tektronic
`aahex`	*American Automation* symbolic HEX file
`mot`	*Motorola* S19 HEX file
`ubrof`	UBROF format
`bin`	Binary file
`mcof`	Microchip PIC COFF
`cof`	Common Object File Format
`cod`	Bytecraft COD file format
`elf`	ELF/DWARF file format

2.6.49 `--OUTPUT=type`: Specify Output File Type

This option allows the type of the output file(s) to be specified. If no `--OUTPUT` option is specified, the output file's name will be derived from the first source or object file specified on the command line.

The available output file format are shown in Table 2.12. More than one output format may be specified by supplying a comma-separated list of tags. Those output file types which specify library formats stop the compilation process before the final stages of compilation are executed. Hence specifying an output file format list containing, e.g. `lib` or `all` will over-ride the non-library output types, and only the library file will be created.

2.6.50 `--PASS1`: Compile to P-code

The `--PASS1` option is used to generate a p-code intermediate files (`.p1` file) from the parser, then stop compilation. Such a file needs to be generated if creating a p-code library file.

2.6.51 `--PRE`: Produce Preprocessed Source Code

The `--PRE` option is used to generate preprocessed C source files with an extension `.pre`. This may be useful to ensure that preprocessor macros have expanded to what you think they should. Use of this option can also create C source files which do not require any separate header files. This is useful when sending files for technical support.

If you wish to see the preprocessed source for the `printf` family of functions, do *not* use this option. The source for this function is customised by the compiler, but only after the code generator has scanned the project for printf usage. Thus, as the –PRE option stops compilation after the preprocessor stage, the code generator will not execute and no printf code will be processed. If this option is omitted, the preprocessed source for `printf` will be retained in the file `doprnt.pre`.

If you wish to see the preprocessed source for the `printf` family of functions, do *not* use this option. The source for this function is customised by the compiler, but only after the code generator has scanned the project for printf usage. Thus, as the –PRE option stops compilation after the preprocessor stage, the code generator will not execute and no printf code will be processed. If this option is omitted, the preprocessed source for `printf` will be retained in the file `doprnt.pre`.

2.6.52 --PROTO: Generate Prototypes

The `--PROTO` option is used to generate `.pro` files containing both ANSI and K&R style function declarations for all functions within the specified source files. Each `.pro` file produced will have the same base name as the corresponding source file. Prototype files contain both ANSI C-style prototypes and old-style C function declarations within conditional compilation blocks.

The extern declarations from each `.pro` file should be edited into a global header file which is included in all the source files comprising a project. The `.pro` files may also contain static declarations for functions which are local to a source file. These static declarations should be edited into the start of the source file. To demonstrate the operation of the `--PROTO` option, enter the following source code as file `test.c`:

```
#include <stdio.h>
add(arg1, arg2)
int *   arg1;
int *   arg2;
{
        return *arg1 + *arg2;
}

void printlist(int * list, int count)
{
        while (count--)
                printf("%d ", *list++);
        putchar('\n');
}
```

If compiled with the command:

```
PICC18 --CHIP=18F242 --PROTO test.c
```

PICC18 will produce `test.pro` containing the following declarations which may then be edited as necessary:

```
/* Prototypes from test.c */
/* extern functions - include these in a header file */
#if     PROTOTYPES
extern int add(int *, int *);
extern void printlist(int *, int);
#else         /* PROTOTYPES */
extern int add();
extern void printlist();
#endif        /* PROTOTYPES */
```

2.6.53 `--RAM=lo-hi,<lo-hi,...>`: Specify Additional RAM Ranges

This option is used to specify memory, in addition to any RAM specified in the chipinfo file, which should be treated as available RAM space. Strictly speaking, this option specifies the areas of memory that may be used by writable (RAM-based) objects, and not necessarily those areas of memory which contain physical RAM. The output that will be placed in the ranges specified by this option are typically variables that a program defines.

Some chips have an area of RAM that can be remapped in terms of its location in the memory space. This, along with any fixed RAM memory defined in the chipinfo file, are grouped an made available for RAM-based objects.

For example, to specify an additional range of memory to that present on-chip, use:

```
--RAM=default,+100-1ff
```

for example. To only use an external range and ignore any on-chip memory, use:

```
--RAM=0-ff
```

This option may also be used to reserve memory ranges already defined as on-chip memory in the chipinfo file. To do this supply a range prefixed with a *minus* character, -, for example:

```
--RAM=default,-100-103
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 103h for allocation of RAM objects.

This option is also used to specify RAM for far objects on PIC18 devices. These objects are stored in the PIC18 extended memory. Any additional memory specified with this option whose address is above the on-chip program memory is assumed to be extended memory implemented as RAM.

For example, to indicate that RAM has been implemented in the extended memory space at addresses 0x20000 to 0x20fff, use the following option.

```
--RAM=default,+20000-20fff
```

See Section 2.7 for use of this option in MPLAB IDE.

2.6.54 `--ROM=lo-hi,<lo-hi,...>|tag`: Specify Additional ROM Ranges

This option is used to specify memory, in addition to any ROM specified in the chip configuration file, which should be treated as available ROM space. Strictly speaking, this option specifies the areas of memory that may be used by read-only (ROM-based) objects, and not necessarily those areas of memory which contain physical ROM. The output that will be placed in the ranges specified by this option are typically executable code and any data variables that are qualified as `const`.

When producing code that may be downloaded into a system via a bootloader the destination memory may indeed be some sort of (volatile) RAM. To only use on-chip ROM memory, this option is not required. For example, to specify an additional range of memory to that on-chip, use:

```
--ROM=default,+100-2ff
```

for example. To only use an external range and ignore any on-chip memory, use:

```
--ROM=100-2ff
```

This option may also be used to reserve memory ranges already defined as on-chip memory in the chip configuration file. To do this supply a range prefixed with a *minus* character, -, for example:

```
--ROM=default,-100-1ff
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 1ffh for allocation of ROM objects.

See Section 2.7 for use of this option in MPLAB IDE.

Table 2.13: Runtime environment suboptions

Suboption	Controls	On (+) implies
init	The code present in the startup module that copies the idata, ibigdata and ifardata psects' ROM-image to RAM.	The idata, ibigdata and ifardata psects' ROM image is copied into RAM.
clib	The inclusion of library files into the output code by the linker.	Library files are linked into the output.
clear	The code present in the startup module that clears the bss, bigbss, rbss and farbss psects.	The bss, bigbss, rbss and farbss psects are cleared.
config	Program unspecified configuration words with a default value	Unspecified configuration words will have a default value programmed.
download	Conditioning of the Intel hex file for use with bootloaders.	Data records in the Intel hex file are padded out to 16 byte lengths and will align on 16 byte boundaries. Startup code will not assume reset values in certain registers.
keep	Whether the start-up module source file is deleted after compilation.	The start-up module is not deleted.
no_startup	Whether the startup module is linked in with user-defined code.	The start-up module is generated and linked into the program.
stackwarn	Checking the depth of the stack used.	The stack depth is monitored at compiled time and a warning will be produced if a potential stack overflow is detected.
plib	Compiler links the Microchip compatible peripheral libraries. Other than <htc.h> no other header files need to be included to use the functions in these libraries. By default this option is not set.	Compiler links the Microchip compatible peripheral libraries.

2.6.55 --RUNTIME=*type*: Specify Runtime Environment

The --RUNTIME option is used to control what is included as part of the runtime environment. The runtime environment encapsulates any code that is present at runtime which has not been defined by the user, instead supplied by the compiler, typically as library code.

All runtime features are enabled by default and this option is not required for normal compilation. The usable suboptions include those shown in Table 2.13.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.56 --SCANDEP: Scan for Dependencies

When this option is used, a .dep (dependency) file is generated. The dependency file lists those files on which the source file is dependant. Dependencies result when one file is #included into another.

2.6.57 --SERIAL=*hexcode*@*address*: Store a Value at this Program Memory Address

This option allows a hexadecimal code to be stored at a particular address in program memory. A typical application for this option might be to position a serial number in program memory. The byte-width of data to store is determined by the byte-width of the hexcode parameter in the option.

A label __serial0 is defined by the runtime startup code that marks the position of the hexadecimal code. This symbol may be referenced by C or assembly code in the usual way.

For example, to store the one byte value, 0, at program memory address 1000h, use the option --SERIAL=*00*@1000. Use the option --SERIAL=*00000000*@1000 to store the same value as a four byte quantity. This option is functionally identical to the corresponding hexmate option. For more detailed information and advanced controls that can be used with this option, refer to Section 5.14.1.15 of this manual.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.58 --SETOPTION=*app*,*file*: Set The Command-line Options for Application

This option is used to supply alternative command line options for the named application when compiling. The *app* component specifies the application that will recieve the new options. The *file* component specifies the name of the file that contains the additional options that will be passed to the application. This option is not required for most projects. If specifying more than one option to a component, each option must be entered on a new line in the option file.

This option can also be used to remove an application from the build sequence. If the *file* parameter is specified as *off*, execution of the named application will be skipped. In most cases this is not desirable as almost all applications are critical to the success of the build process. Disabling a critical application will result in catastrophic failure. However it is permissible to skip a non-critical application such as *clist* or *hexmate* if the final results are not reliant on their function.

2.6.59 --SHROUD: Obfuscate p-code Files

This option should be used in situations where either p-code files or p-code libraries are to be distributed and are built from confidential source code.

C comments, which are normally included into these files, as well as line numbers and variable name will be removed or obfuscated so that the original source code cannot be reconstructed from distributed files.

2.6.60 --STRICT: Strict ANSI Conformance

The --STRICT option is used to enable strict ANSI conformance of all special keywords. HI-TECH C supports various special keywords (for example the persistent type qualifier). If the --STRICT option is used, these keywords are changed to include two *underscore* characters at the beginning of the keyword (e.g. __persistent) so as to strictly conform to the ANSI standard. Be warned that use of this option may cause problems with some standard header files (e.g. <intrpt.h>).

2.6.61 --SUMMARY=*type*: Select Memory Summary Output Type

Use this option to select the type of memory summary that is displayed after compilation. By default, or if the mem suboption is selected, a memory summary is shown. This shows the total memory usage for all memory spaces.

A psect summary may be shown by enabling the psect suboption. This shows individual psects, after they have been grouped by the linker, and the memory ranges they cover. Table 2.14 shows what summary types are available.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.62 --TIME: Report time taken for each phase of build process

Adding --TIME when building generate a summary which shows how much time each stage of the build process took to complete.

Table 2.14: Memory Summary Suboptions

Suboption	Controls	On (+) implies
psect	Summary of psect usage.	A summary of psect names and the addresses they were linked at will be shown.
mem	General summary of memory used.	A concise summary of memory used will be shown.
class	Summary of class usage.	A summary of all classes in each memory space will be shown.
hex	Summary of address used within the hex file.	A summary of addresses and hex files which make up the final output file will be shown.
file	Whether summary information is shown on the screen or shown and saved to a file.	Summary information will be shown on screen and saved to a file.

2.6.63 --VER: Display The Compiler's Version Information

The --VER option will display what version of the compiler is running.

2.6.64 --WARN=*level*: Set Warning Level

The --WARN option is used to set the compiler warning level. Allowable warning levels range from -9 to 9. The warning level determines how pedantic the compiler is about dubious type conversions and constructs. The higher the warning level, the more important the warning message. The default warning level is 0 and will allow all normal warning messages.

Use this option with care as some warning messages indicate code that is likely to fail during execution, or compromise portability.

Warning message can be individually disabled with the --MSGDISABLE option, see 2.6.42. See also Section 2.5 for full information on the compiler's messaging system.

See Section 2.7 for use of this option in MPLAB IDE.

2.6.65 --WARNFORMAT=*format*: Set Warning Message Format

This option sets the format of warning messages produced by the compiler. See Section 2.5.4 for more information on this option. For full information on the compiler's messaging system, see Section 2.5.

2.7 MPLAB IDE v8 Universal Toolsuite Equivalents

When compiling under the MPLAB IDE, it is still the compiler's command-line driver that is being executed and compiling the program. The HI-TECH Universal Toolsuite plugin controls the MPLAB IDE build options dialog that is used to access the compiler options, however these graphical controls ultimately adjust the command-line options passed to the command-line driver when compiling. You can see the command-line options being used when building in MPLAB IDE in the Output window.

The following dialogs and descriptions identify the mapping between the MPLAB IDE v8 dialog controls and command-line options. As the toolsuite is universal across all HI-TECH compilers, not all options are applicable for HI-TECH C Compiler for PIC18 MCUs.

If you are using MPLAB IDE X, see Section 2.8.

2.7.1 Directories Tab

The options in this dialog control the output and search directories for some files. See Figure 2.3 in conjunction with the following command line option equivalents.

1. Output directory
   This selection uses the buttons and fields grouped in the bracket to specify an output directory for files output by the compiler.

2. Include Search path
   This selection uses the buttons and fields grouped in the bracket to specify include (header) file search directories. See 2.6.6.

2.7.2 Compiler Tab

The options in this dialog control the aspects of compilation up to code generation. See Figure 2.4 in conjunction with the following command line option equivalents.

1. Define macros
   The buttons and fields grouped in the bracket can be used to define preprocessor macros. See Section 2.6.3.

2. Undefine macros
   The buttons and fields grouped in the bracket can be used to undefine preprocessor macros. See Section 2.6.15.

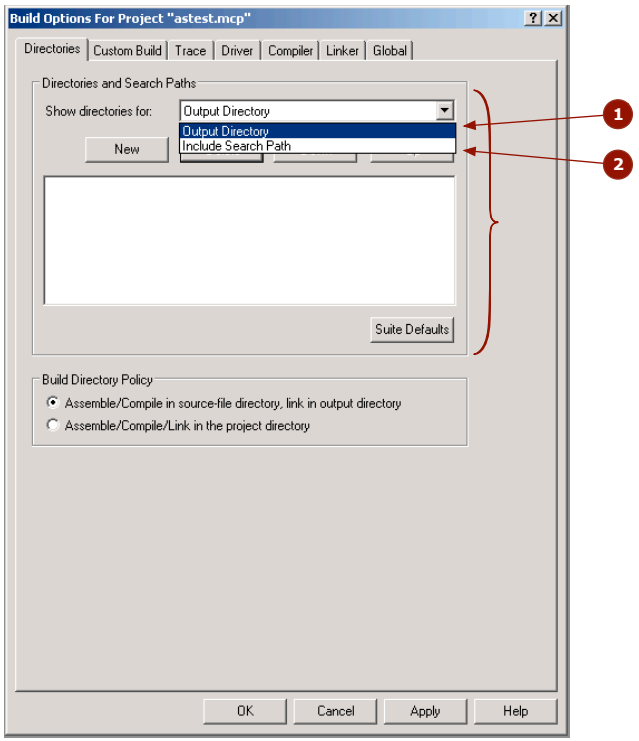


Figure 2.3: The Directories dialog

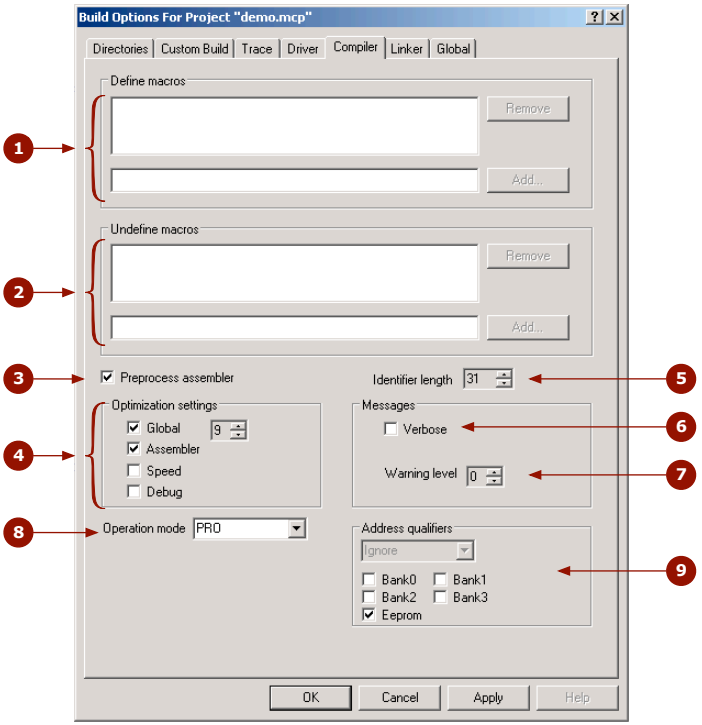


Figure 2.4: The Compiler dialog

3. Preprocess assembly
   This checkbox controls whether assembly source files are scanned by the preprocessor. See Section 2.6.12.

4. Optimization settings
   These controls are used to adjust the different optimizations the compiler employs. See Section 2.6.47.

5. Identifier length
   This selector is currently not implemented. See Section 2.6.10.

6. Verbose
   This checkbox controls whether the full command-lines for the compiler applications are displayed when building. See Section 2.6.16.

7. Warning level
   This selector allows the warning level print threshold to be set. See Section 2.6.64.

8. Operation Mode
   This selector allows the user to force another available operating mode (e.g. Lite, Standard or PRO) other than the default. See Section 2.6.41.

9. Address Qualifier
   This selector allows the user to select the behavior of the address qualifier. See Section 2.6.18.

2.7.3 Linker Tab

The options in this dialog control the link step of compilation. See Figure 2.5 in conjunction with the following command line option equivalents.

1. Runtime options
   These checkboxes control the many runtime features the compiler can employ. See Section 2.6.55.

2. Fill
   This field allows a fill value to be specified for unused memory locations. See Section 2.6.33.

3. Codeoffset
   This field allows an offset for the program to be specified. See Section 2.6.24.

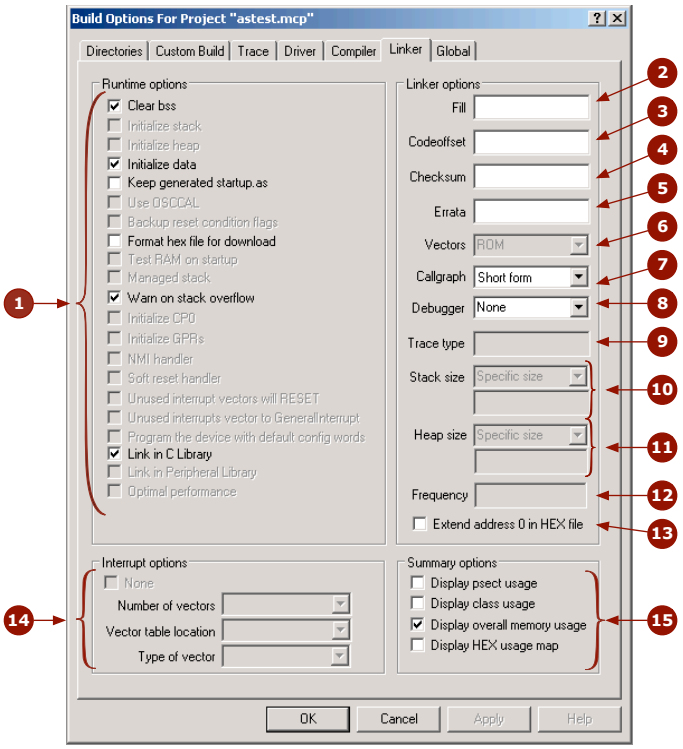


Figure 2.5: The Linker dialog

4. Checksum
   This field allows the checksum specification to be specified. See Section 2.6.20.

5. Errata
   This field allows the errata workarounds employed by the compiler to be controlled. See Section 2.6.30.

6. Vectors
   Not applicable.

7. Callgraph
   Not applicable.

8. Debugger
   This selector allows the type of hardware debugger to be chosen. See Section 2.6.26.

9. Trace type
   Not yet implemented.

10. Stack size
    Not applicable.

11. Heap size
    Not applicable.

12. Frequency
    Not applicable.

13. Extend address 0 in HEX file
    This option specifies that the intel HEX file should have initialization to zero of the upper address. See Section 2.6.49.

14. Interrupt options
    Not applicable.

15. Report Options
    These checkboxes control which summaries are printed after compilation. See Section 2.6.61.

16. Create HTML Files
    This checkbox enables the produced of a web page page that has information relating to the compilation process. It is accessible from `html/index.html` in the output directory. See Section 2.6.37.

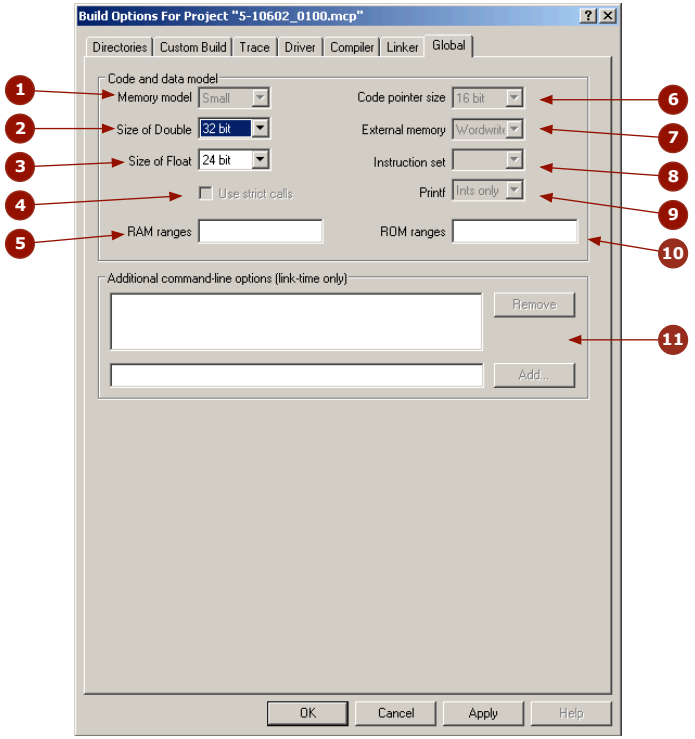


Figure 2.6: The Global dialog

2.7.4 Global Tab

The options in this dialog control aspects of compilation that are applicable throughout code generation and link steps. See Figure 2.6 in conjunction with the following command line option equivalents.

1. Memory model
   Not applicable.

2. Size of Double
   This selector allows the size of `double` types to be selected. See Section 2.6.27.

3. Size of Float
   This selector allows the size of `float` types to be selected. See Section 2.6.34.

4. Use strict calls
   Not applicable.

5. RAM ranges
   This field allows the default RAM (data space) memory used to be adjusted. See Section 2.6.53.

6. Code pointer size
   Not applicable.

7. External memory
   This selector allows the type of external memory access to be specified. See Section 2.6.29.

8. Instruction set
   Not applicable.

9. Printf
   Not applicable.

10. ROM ranges
    This field allows the default ROM (program space) memory used to be adjusted. See Section 2.6.54.

11. Additional Command-line options
    These widgets allow options which have no direct widget in the Build Options dialog to be specified. The options entered here are only used during the second phase of compilation—the code generation and link steps—and will not affect the preprocessing or parsing compilation steps. These options must be compiler driver options, as described by Section 2.6.

2.8 MPLAB X Universal Toolsuite Equivalents

When compiling under the MPLAB X IDE, it is still the compiler's command-line driver that is being executed and compiling the program. The HI-TECH compiler plugins controls the MPLAB X IDE Properties dialog that is used to access the compiler options, however these graphical controls ultimately adjust the command-line options passed to the command-line driver when compiling. You can see the command-line options being used when building in MPLAB X IDE in the Output window.

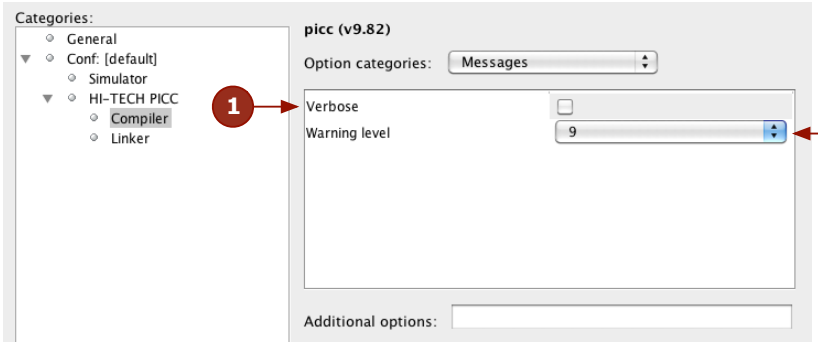


Figure 2.7: The Messages category

The following dialogs and descriptions identify the mapping between the MPLAB X IDE dialog controls and command-line options. As the plugin is universal across all HI-TECH compilers, not all options are applicable for HI-TECH C Compiler for PIC18 MCUs.

If you are using MPLAB IDE v8, see Section 2.7.

2.8.1 Compiler Category

The panels in this category control aspects of compilation of C source.

2.8.1.1 Messages

These options relate to messages produced by the compiler (see Section 2.5 for more information). See Figure 2.7 in conjunction with the following command line option equivalents.

1. Verbose:
   This checkbox controls whether the full command-lines for the compiler applications are displayed when building. See Section 2.6.16.

2. Warning level:
   This selector allows the warning level print threshold to be set. See Section 2.6.64.

2.8.1.2 Address Qualifiers

This option illustrated in Figure 2.8 relates to how the compiler responds to some qualifiers.

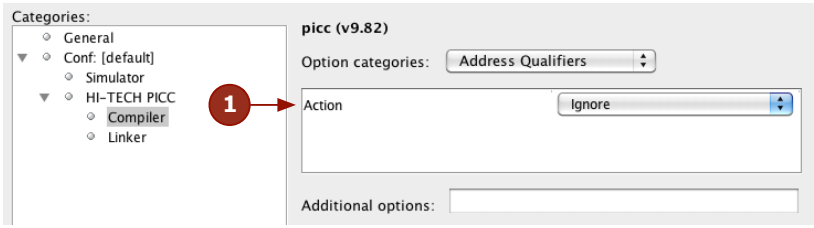


Figure 2.8: The Address Qualifiers category

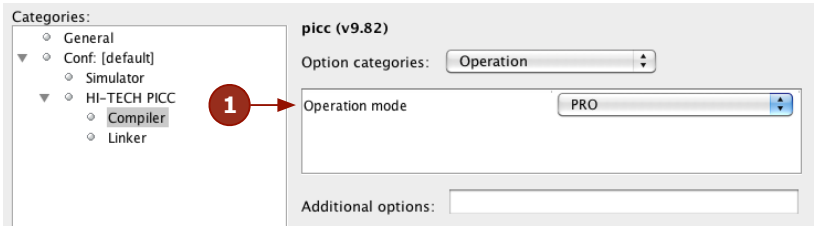


Figure 2.9: The Operation Category

1. Address Qualifier:
   This selector allows the user to select the behavior of the address qualifiers. See Section 2.6.18.

2.8.1.3 Operation

This option illustrated in Figure 2.9 relates to the operating mode of the compiler.

1. Operation Mode:
   This selector allows the user to force another available operating mode (e.g. Lite or PRO) other than the default. See Section 2.6.41.

2.8.1.4 Preprocessor

These options, shown in Figure 2.10, relate to preprocessor operation.

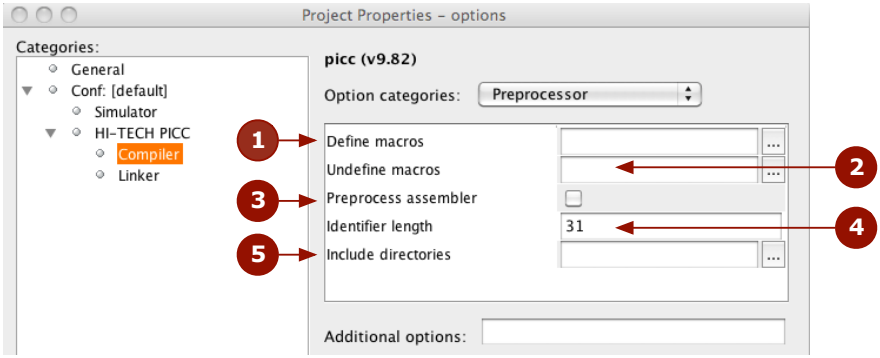


Figure 2.10: The Preprocessor Category

1. Define macros:
   The buttons and fields grouped in the bracket can be used to define preprocessor macros. See Section 2.6.3.

2. Undefine macros:
   The buttons and fields grouped in the bracket can be used to undefine preprocessor macros. See Section 2.6.15.

3. Preprocess assembly:
   This checkbox controls whether assembly source files are scanned by the preprocessor. See Section 2.6.12.

4. Identifier length:
   This selector is currently not implemented. See Section 2.6.10.

5. Include Directories:
   This selection uses the buttons and fields grouped in the bracket to specify include (header) file search directories. See Section 2.6.6.

2.8.1.5 Optimization

These options, shown in Figure 2.11, relate to optimizations performed by the compiler.

1. Optimization set:
   This controls enables different optimizations the compiler employs. See Section 2.6.47.

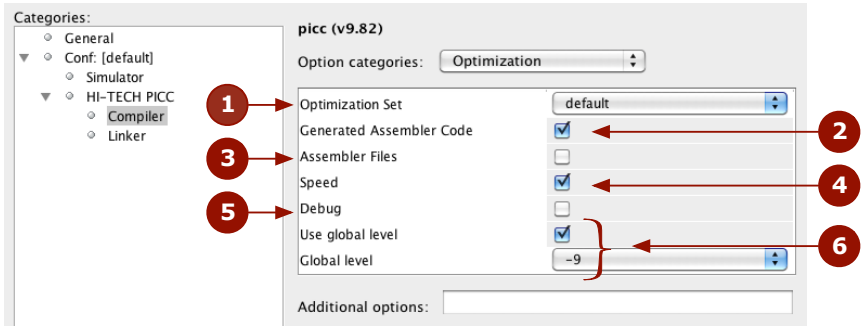


Figure 2.11: The Optimization Category

2. Generated Assembly Code:
   The control enables optimization of assembly code generated from C code. See Section 2.6.47.

3. Assembly Files:
   This control enables optimization of assembly source files. See Section 2.6.47.

4. Speed:
   This control allows you to toggle between speed- or space-biased optimizations. See Section 2.6.47.

5. Debug:
   This control allows you to disable some optimizations so that generated code is better behaved in debuggers. See Section 2.6.47.

6. Global Level:
   This control allows you to enable the global C optimizer and adjust the optimization level. See Section 2.6.47.

2.8.2 Linker Category

The options in this dialog control the aspects of the second stage of compilation including code generation and linking.

2.8.2.1 Data

These options, shown in Figure 2.12 relate to C data types and data memory.

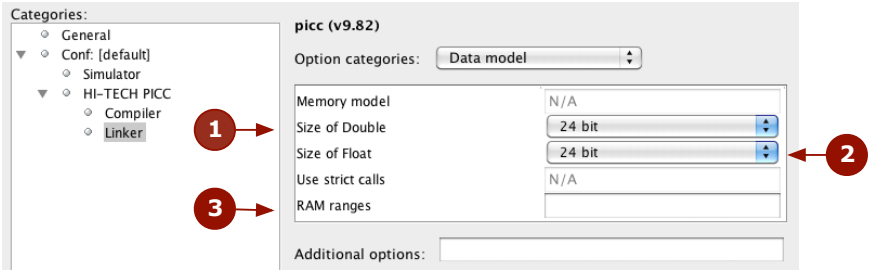


Figure 2.12: The Data Category

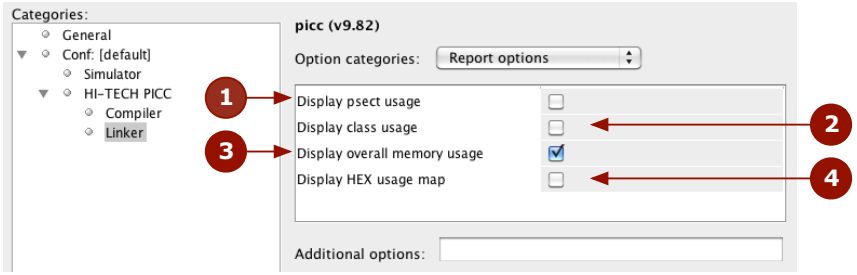


Figure 2.13: The Report Category

1. Size of Double:
   This selector allows the size of the `double` type to be selected. See Section 2.6.27.

2. Size of Float:
   This selector allows the size of the `float` type to be selected. See Section 2.6.34.

3. RAM ranges:
   This field allows the default RAM (data space) memory used to be adjusted. See Section 2.6.53.

2.8.2.2 Report

These options, shown in Figure 2.13 relate to information printed after compilation.

1. Display Psect Usage:
   This checkbox enables printing of psect locations after compilation. See Section 2.6.61.

2. Display Class Usage:
   This checkbox enables printing of psect ranges sorted by class after compilation. See Section 2.6.61.

3. Display overall memory usage:
   This checkbox enables printing of a memory summary after compilation. See Section 2.6.61.

4. Display HEX Usage:
   This checkbox enables printing of a graphical representation of HEX file contents after compilation. See Section 2.6.61.

2.8.2.3 Runtime

All the widgets in Figure 2.14 correspond to suboptions of the –RUNTIME option, see Section 2.6.55.

2.8.2.4 Code

These options, shown in Figure 2.15 relate to program memory.

1. External memory:
   The control allows configuration of the external memory interface. See Section 2.6.29.

2. ROM ranges:
   This field allows the default ROM (program space) memory used to be adjusted. See Section 2.6.54.

2.8.2.5 Additional

These options, shown in Figure 2.16 relate to miscellaneous options.

1. Extra linker Options:
   This field allows specification of additional user-defined linker options, see Section 2.6.8.

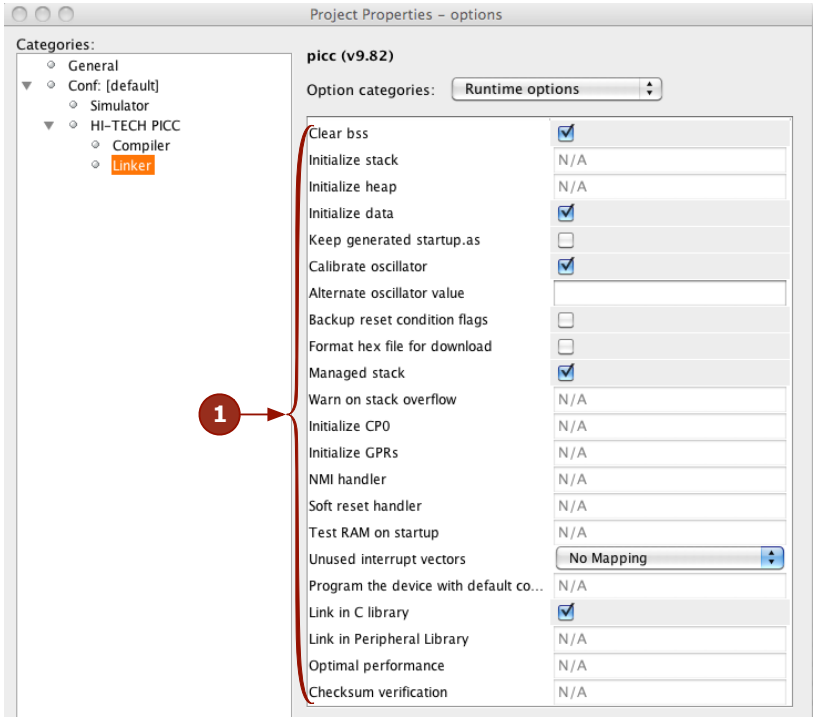


Figure 2.14: The Runtime Category

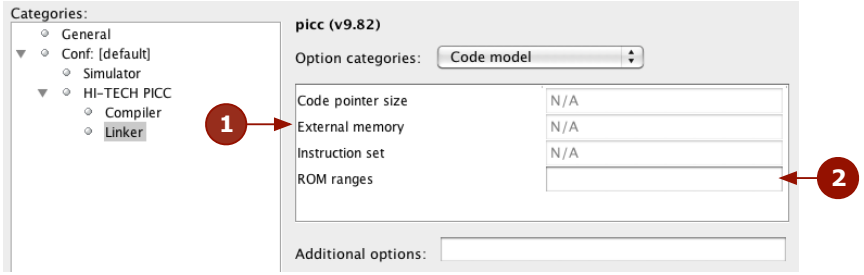


Figure 2.15: The Code Category

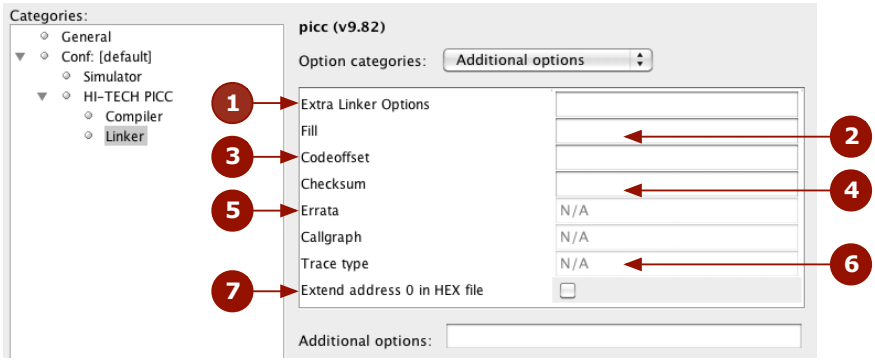


Figure 2.16: The Additional Category

2. Fill:
   This field allows a fill value to be specified for unused memory locations. See Section 2.6.33.

3. Codeoffset:
   This field allows an offset for the program to be specified. See Section 2.6.24.

4. Checksum:
   This field allows the checksum specification to be specified. See Section 2.6.20.

5. Errata:
   This allows customization of the errata workarounds applied by the compiler. See Section 2.6.30.

6. Trace type:
   Not yet implemented.

7. Extend address 0 in HEX file:
   This option specifies that the intel HEX file should have initialization to zero of the upper address. See Section 2.6.49.

Chapter 3

C Language Features

HI-TECH C Compiler for PIC18 MCUs supports a number of special features and extensions to the C language which are designed to ease the task of producing ROM-based applications. This chapter documents the compiler options and special language features which are specific to the *Microchip* PIC 18 family of processors.

3.1 ANSI Standard Issues

3.1.1 Divergence from the ANSI C Standard

HI-TECH C Compiler for PIC18 MCUs diverges from the ANSI C standard in one area: function recursion.

Due to the PIC18's hardware limitations of no easily-usable stack and limited memory, function recursion is unsupported.

3.1.2 Implementation-defined behaviour

Certain sections of the ANSI standard have implementation-defined behaviour. This means that the exact behaviour of some C code can vary from compiler to compiler. Throughout this manual are sections describing how the HI-TECH C Compiler for PIC18 MCUs compiler behaves in such situations.

3.1.3 Non-ANSI Operations

HI-TECH C Compiler for PIC18 MCUscan detect specific sequences of portable ANSI C code that implicitly implements a rotate operation. The C language only specifies a left and right shift operator, but no rotate operator. The code generator encodes matching sequences using assembly rotate instructions where possible.

The code sequence to implement a rotate right by 1 bit looks like:

```
var = (var >> 1) | (var << 7);
```

where `var` must be an `unsigned char` or:

```
var = (var >> 1) | (var << 15);
```

where `var` must be an `unsigned int`. Rotates can be either left or right and of any number of bits. Note that a rotate left of 1 bit is equivalent to a rotate right of 7 bits when dealing with byte-wide variables, or a rotate right of 15 bits when dealing with 2-byte quantities.

3.1.4 C18 Compatibility

HI-TECH C Compiler for PIC18 MCUs provides ANSI C compliance, as well as HI-TECH C specific extensions to the C language. In addition, the compiler can be placed into a special mode that will allow it to accept language extensions used in the MPLAB C Compiler for PIC18 MCUs (former called, and referenced here as, “MPLAB C18” or just “C18”) and so can be used as a replacement for this compiler. This mode, which is called C18 compatibility mode, allows HI-TECH C Compiler for PIC18 MCUs to compile legacy projects that were designed for the MPLAB compiler. When not in this mode, the compiler will only accept ANSI C or the HI-TECH C language extensions.

Support for compatibility in version 9.80 of HI-TECH C Compiler for PIC18 MCUs is beta only. Not all syntax may be supported and compliance with the C18 operation is not guaranteed.

In C18 compatibility mode, the compiler will accept most C language extensions offered by MPLAB C18 and many of the directives and instruction formats accepts by the internal C18 assembler. Separate assembly modules, that would normally be passed to MPASM, and the directives and syntax that these files can contain, are not supported.

To provide further compatibility of legacy projects, a replacement MPLAB C18 driver (`mcc18`) is provided HI-TECH C Compiler for PIC18 MCUs that mimics the operation of the MPLAB C18

compiler executable. When executed, this replacement driver will automatically invoke the HI-TECH compiler in the compatibility mode and transcode options to the HI-TECH equivalent. This allows command line options, batch files, linker scripts or MPLAB IDE projects setup for MPLAB C18 to use the HI-TECH compiler with virtually no modification. Legacy projects need only be associated with, and call, the replacement MPLAB C18 driver to build using the HI-TECH compiler in compatibility mode.

This manual only describes the operation of the compiler in HI-TECH compatibility mode. For the meaning of language extensions and compiler operation in C18 compatibility mode, refer to the MPLAB C Compiler for PIC18 MCUs compiler manual.

When you run the installer for this compiler, you have the option of converting your existing MPLAB IDE projects configured to use the MPLAB C Compiler for PIC18 MCUs to the HI-TECH compiler. Once converted, these projects will use the HI-TECH compiler in compatibility mode and the HI-TECH replacement compiler driver (`mcc18`) will be called. At any time, you may re-run the compiler's activation program from the Windows Start menu to convert projects if you did not perform this action when installing.
If you convert your C18 projects, this will mean that they will not use the installed MPLAB C Compiler for PIC18 MCUs unless you revert the Location paths of the executables in the Select Language Toolsuite dialog in the MPLAB IDE (v8), or the Built Tool settings in the Embedded tab of the MPLAB X IDE Preferences dialog.

It is recommended that for new projects, the HI-TECH syntax is followed.

3.2 Processor-related Features

HI-TECH C Compiler for PIC18 MCUs has many features which relate directly to the PIC18 family of processors. These are detailed in the following sections.

3.2.1 Processor Support

HI-TECH C Compiler for PIC18 MCUs supports the full range of Microchip PIC 18 processors. However, new devices in this family are frequently released. There are several ways you can check if the compiler you are using supports a particular device. From MPLAB IDE, open the Build Options dialog. Select the Driver tab. In the Available Drivers field, select the compiler you wish to use. A list of all devices supported by that compiler will be shown in the Selected Driver Information and Supported Device area, towards the center of the dialog.

From the command line, the same information can be obtained. Run the compiler you wish to use and pass it the option `--CHIPINFO` (See Section 2.6.22). A list of all devices will be printed.

Additional code-compatible processors may be added by editing the `picc-18.ini` file in the DAT directory. User-defined processors should be placed at the end of the file. The header of the file explains how to specify a processor. Newly added processors will be available the next time you compile by selecting the name of the new processor on the command line in the usual way.

3.2.2 Device Header Files

There is one header file that is recommended be included into each source file you write. The file is `<htc.h>` and is a generic file that will include other device and chip-specific header files when you build your project.

Inclusion of this file will allow access to SFRs via special variables, as well as macros which allow special memory access or inclusion of special instructions, like `CLRWDT()`.

If you are writing assembly code, there are different header files that define assembly symbols that represent the SFRs. See 3.10.3.1 for more information on these files.

3.2.3 Stack

The hardware stack on PIC18 devices is limited in depth and cannot be manipulated directly. It is only used for function return address and cannot be used for program data. The compiler implements a compiled stack for local data objects, see Section 3.4.1.1 for information on how this is achieved.

You must ensure that the maximum stack depth is not exceeded; otherwise, code may fail. Calling too many nested functions may overflow the stack, and it is important to take into account interrupts, which also use levels of the stack.

A call graph is provided by the code generator in the assembler list file. This will indicate the stack levels at each function call and can be used as a guide to stack depth. The code generator may also produce warnings if the maximum stack depth is exceeded, see Section 2.6.55.

Both of these are guides to stack usage. Optimizations and the use of interrupts can change the stack depth used by a program over that determined by the compiler.

3.2.4 Configuration Fuses

The PIC18 processor's have several locations which contain the *configuration bits* or *fuses*. These bits may be set using the configuration pragma. The pragma has the forms:

```
#pragma config setting = state or value
#pragma config register = value
```

where *setting* is a configuration setting descriptor, e.g. WDT, and *state* is a textual description of the desired state, e.g. OFF. The *value* field is a numerical value that can be used in preference to a descriptor. The value is assigned to the setting. For example,

```
#pragma config WDT = ON       // turn on watchdog timer
#pragma config WDT = 1        // an alternate form of the above
#pragma config WDTPS = 0x1A   // specify the watchdog timer postscale value
```

One pragma can be used to program several settings by separating each setting-value pair with a comma. For example, the above could be specified with one pragma, as in the following.

```
#pragma config WDT=ON, WDTPS = 0x1A
```

Rather than specify individual settings, the entire *register* may be programmed with one numerical *value*, if you prefer, e.g.

```
#pragma config CONFIG1L = 0x8F
```

The settings and values associated with each device can be determined from an HTML guide. Open the file chipinfo.html, which is located in the DOCS directory of your compiler installation. Click on your target device and it will show you the settings and values that are appropriate with this pragma. Check your device datasheet for more information.

The compiler also has legacy support for the __CONFIG and __PROG_CONFIG macros which allow configuration bit symbols or a configuration word value, respectively, to be specified. For example:

```
__CONFIG(2, BW8 & PWRTDIS & WDTPS1 & WDTEN);  // specify symbols
```

or

```
__PROG_CONFIG(1, 0xFE57);  // specify a literal constant value
```

You cannot use the symbols in the __PROG_CONFIG macro, nor can you use a literal value in the __CONFIG macro.

Use the pragma in preference to the macros for new projects. To use these macros, ensure you include <htc.h> in your source file. Symbols for the macros can be found in the .cfgdata fies contained in the dat/cfgdata directory of your compiler installation directory.

Neither the config pragma, nor the macros, produce executable code and so should ideally be placed outside function definitions.

3.2.5 ID Locations

Some PIC18 devices have locations outside the addressable memory area that can be used for storing program information, such as an ID number. The `config` pragma may also be used to place data into these locations by using a special register name. The pragma is used in a manner similar to:

```
#pragma config IDLOCX = value
```

where *X* is the number (position) of the ID location, and *value* is the nibble or byte which is to be positioned into that ID location. If the value is larger than the maximum value allowable for each location on the target device, the value will be truncated and a warning message issued. The size of each ID location value varies from device to device. See your device datasheet for more information. For example:

```
#pragma config IDLOC0 = 1
#pragma config IDLOC1 = 4
```

will attempt fill the first two ID locations with 1 and 4. One pragma can be used to program several locations by separating each register-value pair with a comma. For example, the above could also be specified as:

```
#pragma config IDLOC0 = 1, IDLOC1 = 4
```

The compiler also has legacy support for the `__IDLOC` macro, for example:

```
__IDLOC(15F01);
```

To use this macro, ensure you include `<htc.h>` in your source file.

Neither the `config pragma`, nor the `__IDLOC` macro, produce executable code and so should ideally be placed outside function definitions.

3.2.6 Bit Instructions

Wherever possible, HI-TECH C Compiler for PIC18 MCUs will attempt to use the PIC18 bit instructions. For example, when using a bitwise operator and a mask to alter a bit within an integral type, the compiler will check the mask value to determine if a bit instruction can achieve the same functionality.

```
unsigned int foo;
foo |= 0x40;
```

will produce the instruction:

```
bsf _foo,6
```

To set or clear individual bits within integral type, the following macros could be used:

```
#define bitset(var,bitno)     ((var) |= 1UL << (bitno))
#define bitclr(var,bitno)     ((var) &= ~(1UL << (bitno)))
```

To perform the same operation as above, the bitset macro could be employed as follows:

```
bitset(foo,6);
```

3.2.7 EEPROM and Flash Runtime Access

The compiler offers several methods of accessing EEPROM and Flash memory. These are described in the following sections.

3.2.7.1 EEPROM Access

For those PIC devices that support external programming of their EEPROM data area, the `__EEPROM_DATA()` macro can be used to place the initial EEPROM data values into the HEX file ready for programming. The macro is used as follows.

```
#include <htc.h>
__EEPROM_DATA(0, 1, 2, 3, 4, 5, 6, 7);
```

The macro accepts eight parameters, being eight data values. Each value should be a byte in size. Unused values should be specified as a parameter of zero. The macro may be called multiple times to define the required amount of EEPROM data. It is recommended that the macro be placed outside any function definitions.

The macro defines, and places the data within, a psect called `eeprom_data`. This psect is positioned by a linker option in the usual way. This macro is not used to write to EEPROM locations during run-time.

There are functions to access the EEPROM at runtime. These are provided by the peripheral library.

The function forms of these routines use the prototypes:

```
unsigned char eeprom_read(unsigned int addr);
```

Read and return the byte location at `addr` in the EEPROM memory.

```
void eeprom_write(unsigned int addr, unsigned char value);
```

Write `value` to the EEPROM memory at the address `addr`.

The macros `EEPROM_READ()` and `EEPROM_WRITE()` are for legacy support, but actually call the function versions of these routines. So to write a value, the following will call `eeprom_write`.

```
EEPROM_WRITE(address, value);
```

To read a byte of data from an address in EEPROM memory, and store it in a variable:

```
variable=EEPROM_READ(address);
```

For convenience, `__EEPROMSIZE` predefines the number of bytes of EEPROM available on chip.

3.2.7.2 Flash Access

Routines to access the flash memory are provided in the peripheral libraries. See 2.6.55 for information on how these can be specified when building.

The prototypes for the available functions are:

```
void EraseFlash(unsigned long startaddr, unsigned long endaddr);
```

Erase flash memory from `startadd` to `endaddr`.

```
void ReadFlash(unsigned long startaddr, unsigned int num_bytes,
               unsigned char *flash_array);
```

Read `num_bytes` bytes of flash memory starting from `startaddr`, storing them in the array specified by `flash_array`.

```
void WriteBytesFlash(unsigned long startaddr, unsigned int num_bytes,
                     unsigned char *flash_array);
```

Write `num_bytes` bytes of data from `flash_array` into flash memory starting at address `start_addr`.

```
void WriteWordFlash(unsigned long startaddr, unsigned int data);
```

Write `data` into flash at address `start_addr`.

```
void WriteBlockFlash(unsigned long startaddr, unsigned char num_blocks,
                     unsigned char *flash_array);
```

Write num_blocks blocks of data from flash_array into flash memory starting at address start_addr. The block size is device dependent and is indicated in the device datasheet.

Note that when flash memory is written, the entire block that contains the new values must be erased and then written as a whole. You need to ensure that the start and end address you specify in these routines take these boundaries into account. If you only wish to write some of the locations in the block, then you must read in the block and store it in a RAM array, modify the copy to include the changes required, then write then modifed array back to flash. Similarly, when erasing flash memory, you should erase entire blocks. Check your device datasheet for flash block sizes and exact operation.

3.2.8 Using SFRs From C Code

The Special Function Registers (SFRs) are registers which control aspects of the MCU operation or that of peripheral modules on the device. Most of these registers are memory mapped, which means that they appear at specific addresses in the data memory space of the device. With some registers, the bits within the register control independent features. Some registers are read-only; some are write-only.

Memory-mapped SFRs are accessed by special C variables that are placed at the addresses of the registers. Variables that are placed at specific addresses are called absolute variables and are described in Section 3.4.2. These variables can be accessed like any ordinary C variable so no special syntax is required to access SFRs. Bit variables, as well as structures (with bit-fields), can also be made absolute and so either can be used to represent bits within the register.

The SFR variables are predefined in header files and will be accessible once the <htc.h> header file (see Section 3.2.2) has been included into your source code. Both bit variables and structures with bit-fields are defined by the inclusion of this header file so you may use either in your source code.

The names given to the C variables, which map over the registers and bit variables, or bit-fields, within the registers are based on the names specified in the device data sheet. However, as there can be duplication of some bit names within registers, there may be differences in the nomenclature. The names of the structures that hold the bit-fields will typically be those of the corresponding register followed by bits. For example, the following shows code that includes the generic header file, clears PORTA as a whole, sets bit 0 of PORTA using a bit variable and sets bit 2 of PORTA using the structure/bit-field definitions.

```
#include <htc.h>
void main(void)
{
   PORTA = 0x00;
   RA0 = 1;
```

```
        PORTAbits.RA2 = 1;
    }
```

To confirm the names that are relevant for the device you are using, check the device specific header file that <htc.h> will include for the definitions of each variable. These files will be located in the include directory of the compiler and will have a name that represents the device. There is a one-to-one correlation between device and header file name that will be included by <htc.h>, e.g. when compiling for a PIC18F452 device the <htc.h> header file will include (among other files) <pic18f452.h>.

For compatibility, an additional header file using a C18-style name is also shipped with the compiler. This will #include the "pic" version of the same file. So, for the PIC18F452 device, the file <p18f452.h> can be used for legacy projects.

Care should be taken when accessing some SFRs from C code or from assembly in-line with C code. Some registers are used by the compiler to hold intermediate values of calculations, and writing to these registers directly can result in code failure. The compiler does not detect when SFRs have changed as a result of C or assembly code that writes to them directly. The list of registers used by the compiler and further information can be found in Section 3.7.

SFRs associated with peripherals are not used by the compiler to hold intermediate results and can be changed as you require. Always ensure that you confirm the operation of peripheral modules from the device data sheet.

3.2.8.1 Multi-byte SFRs

Some of the SFRs associated with the PIC18 can be grouped to form multi-byte values, e.g. in the device datasheet the TMR*x*H and TMR*x*L register together form a 16-bit timer count value, TMR*x*. Depending on the device and mode of operation, there may be hardware requirements to read these registers correctly, e.g. the TMR*x*L register often must be read before trying to read the TMR*x*H register to obtain a valid 16-bit result.

Although it is possible to define a word-sized C variable to map over such registers, the order in which HI-TECH C Compiler for PIC18 MCUs the bytes would be read will vary from expression to expression, i.e. it may read the most significant byte first, or the least.

Multi-byte timer registers are not supported by the compiler header files. It highly recommended that the existing SFR definitions for each byte of the timer registers be used. Each SFR should be accessed directly and in the required order by the programmer's code. This will ensure a much higher degree of portability.

The following code copies the two byte registers into C unsigned variable i for subsequent use.

```
i = TMR0L;
i += TMR0H << 8;
```

Table 3.1: Basic data types

Type	Size (bits)	Arithmetic Type
`bit`	1	unsigned integer
`char`	8	signed or unsigned integer
`unsigned char`	8	unsigned integer
`short`	16	signed integer
`unsigned short`	16	unsigned integer
`int`	16	signed integer
`unsigned int`	16	unsigned integer
`long`	32	signed integer
`unsigned long`	32	unsigned integer
`float`	24	real
`double`	24 or 32[1]	real

Macros are also provided to perform common operations, like reading and writing the timer registers, and which read the registers in the correct order. See the macros `READTIMER`*x* and `WRITETIMER`*x* in Chapter A.

3.3 Supported Data Types and Variables

The HI-TECH C Compiler for PIC18 MCUs compiler supports basic data types with 1, 2, 3 and 4 byte sizes. All multi-byte types follow *least significant byte first* format, also known as *little-endian*. Word size values thus have the least significant byte at the lower address, and double word size values have the least significant byte and least significant word at the lowest address. Table 3.1 shows the data types and their corresponding size and arithmetic type.

3.3.1 Radix Specifiers and Constants

The format of integral constants specifies their radix. HI-TECH C Compiler for PIC18 MCUs supports the ANSI standard radix specifiers as well as ones which enables binary constants to specified in C code. The format used to specify the radices are given in Table 3.2. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

Any integral constant will have a type which is the smallest type that can hold the value without overflow. The suffix `l` or `L` may be used with the constant to indicate that it must be assigned either a `signed long` or `unsigned long` type, and the suffix `u` or `U` may be used with the constant to

Table 3.2: Radix formats

Radix	Format	Example
binary	`0b`*number* or `0B`*number*	0b10011010
octal	`0`*number*	0763
decimal	*number*	129
hexadecimal	`0x`*number* or `0X`*number*	0x2F

indicate that it must be assigned an unsigned type, and both `l` or `L` and `u` or `U` may be used to indicate `unsigned long int` type.

Floating-point constants have `double` type unless suffixed by `f` or `F`, in which case it is a `float` constant. The suffixes `l` or `L` specify a `long double` type which is considered an identical type to `double` by HI-TECH C Compiler for PIC18 MCUs.

Character constants are enclosed by single quote characters `'`, for example `'a'`. A character constant has `char` type. Multi-byte character constants are not supported.

String constants or string literals are enclosed by double quote characters `"`, for example `"hello world"`. The type of string constants is `const char []` and the strings are stored in the program memory. Assigning a string constant to a non-`const char` pointer will generate a warning from the compiler. For example:

```
char * cp= "one";          // "one" in ROM, produces warning
const char * ccp= "two";   // "two" in ROM, correct
```

Defining and initializing a non-`const` array (i.e. not a pointer definition) with a string, for example:

```
char ca[]= "two";          // "two" different to the above
```

produces an array in data space which is initialised at startup with the string `"two"` (copied from program space), whereas a constant string used in other contexts represents an unnamed `const`-qualified array, accessed directly in program space.

HI-TECH C will use the same storage location and label for strings that have identical character sequences, except where the strings are used to initialise an array residing in the data space as shown in the last statement in the previous example.

Two adjacent string constants (i.e. two strings separated *only* by white space) are concatenated by the compiler. Thus:

```
const char * cp = "hello " "world";
```

assigned the pointer with the string `"hello world"`.

3.3.2 Bit Data Types and Variables

HI-TECH C Compiler for PIC18 MCUssupports `bit` integral types which can hold the values 0 or 1. Single `bit` variables may be declared using the keyword bit. `bit` objects declared within a function, for example:

```
static bit init_flag;
```

will be allocated in the bit-addressable psect `rbit`, and will be visible only in that function. When the following declaration is used outside any function:

```
bit init_flag;
```

`init_flag` will be globally visible, but located within the same psect.

Bit variables cannot be `auto` or parameters to a function. A function may return a `bit` object by using the `bit` keyword in the functions prototype in the usual way. The bit return value will be returning in the carry flag in the status register.

Bit variables behave in most respects like normal `unsigned char` variables, but they may only contain the values 0 and 1, and therefore provide a convenient and efficient method of storing boolean flags without consuming large amounts of internal RAM. It is, however, not possible to declared pointers to `bit` variables or statically initialise `bit` variables.

Operations on `bit` objects are performed using the single bit instructions (`bsf` and `bcf`) wherever possible, thus the generated code to access `bit` objects is very efficient.

Note that when assigning a larger integral type to a `bit` variable, only the least-significant bit is used. For example, if the `bit` variable `bitvar` was assigned as in the following:

```
int data = 0x54;
bit bitvar;
bitvar = data;
```

it will be cleared by the assignment since the least significant bit of `data` is zero. If you want to set a bit variable to be 0 or 1 depending on whether the larger integral type is zero (false) or non-zero (true), use the form:

```
bitvar = data != 0;
```

The psects in which `bit` objects are allocated storage are declared using the `bit PSECT` directive flag. Eight bit objects will take up one byte of storage space which is indicated by the psect's scale value of 8 in the map file. The length given in the map file for bit psects is in units of bits, not bytes. All addresses specified for bit objects are also bit addresses.

The `bit` psects are cleared on startup, but are not initialised. To create a bit object which has a non-zero initial value, explicitly initialise it at the beginning of your code.

If the PICC18 flag `--STRICT` is used, the `bit` keyword becomes unavailable.

3.3.3 Using Bit-Addressable Registers

The bit variable facility may be combined with absolute variable declarations (see Section 3.4.2) to access bits at specific addresses. Absolute bit objects are numbered from 0 (the least significant bit of the first byte) up. Therefore, bit number 3 (the fourth bit in the byte since numbering starts with 0) in byte number 5 is actually absolute bit number 43 (that is 8bits/byte * 5 bytes + 3 bits).

For example, to access the *power down detection flag* bit in the RCON register, declare `RCON` to be a C object at absolute address 03h, then declare a `bit` variable at absolute bit address 27:

```
static unsigned char RCON @ 0xFD0;
static near bit PD @ (unsigned)&RCON*8+2;
```

Note that all standard registers and bits within these registers are defined in the header files provided. The only header file you need to include to have access to the PIC18 registers is `<htc.h>`. At compile time, this will include the appropriate header for the selected chip.

3.3.4 8-Bit Integer Data Types and Variables

HI-TECH C Compiler for PIC18 MCUs supports both `signed char` and `unsigned char` 8-bit integral types. If the `signed` or `unsigned` keyword is absent from the variable's definition, the default type is `unsigned char` unless the PICC18 `--CHAR=signed` option is used, in which case the default type is `signed char`. The `signed char` type is an 8-bit two's complement signed integer type, representing integral values from -128 to +127 inclusive. The `unsigned char` is an 8-bit unsigned integer type, representing integral values from 0 to 255 inclusive. It is a common misconception that the C `char` types are intended purely for ASCII character manipulation. This is not true, indeed the C language makes no guarantee that the default character representation is even ASCII. The `char` types are simply the smallest of up to four possible integer sizes, and behave in all respects like integers.

The reason for the name "char" is historical and does not mean that `char` can only be used to represent characters. It is possible to freely mix `char` values with `short`, `int` and `long` values in C expressions. With HI-TECH C Compiler for PIC18 MCUs the `char` types will commonly be used for a number of purposes, as 8-bit integers, as storage for ASCII characters, and for access to I/O locations.

Variables may be declared using the `signed char` and `unsigned char` keywords, respectively, to hold values of these types. Where only `char` is used in the declaration, the type will be `signed char` unless the option, mentioned above, to specify `unsigned char` as default is used.

3.3.5 16-Bit Integer Data Types

HI-TECH C Compiler for PIC18 MCUs supports four 16-bit integer types. `short` and `int` are 16-bit two's complement signed integer types, representing integral values from -32,768 to +32,767 inclu-

sive. `Unsigned short` and `unsigned int` are 16-bit unsigned integer types, representing integral values from 0 to 65,535 inclusive. All 16-bit integer values are represented in *little endian* format with the least significant byte at the lower address.

Variables may be declared using the `signed short int` and `unsigned short int` keyword sequences, respectively, to hold values of these types. When specifying a `short int` type, the keyword `int` may be omitted. Thus a variable declared as `short` will contain a `signed short int` and a variable declared as `unsigned short` will contain an `unsigned short int`.

3.3.6 24-Bit Integer Data Types

HI-TECH C Compiler for PIC18 MCUs supports two 24-bit integer types. `short long` are 24-bit two's complement signed integer types, representing integral values from -8,388,608 to +8,388,607 inclusive. `Unsigned short long` are 24-bit unsigned integer types, representing integral values from 0 to 16,777,215 inclusive. All 24-bit integer values are represented in *little endian* format with the least significant byte at the lower address.

Variables may be declared using the `signed short long int` and `unsigned short long int` keyword sequences, respectively, to hold values of these types. When specifying a `short long int` type, the keyword `int` may be omitted. Thus a variable declared as `short long` will contain a `signed short long int` and a variable declared as `unsigned short long` will contain an `unsigned short long int`.

3.3.7 32-Bit Integer Data Types and Variables

HI-TECH C Compiler for PIC18 MCUssupports two 32-bit integer types. `Long` is a 32-bit two's complement signed integer type, representing integral values from -2,147,483,648 to +2,147,483,647 inclusive. The `unsigned long` type is a 32-bit unsigned integer type, representing the integral values from 0 to 4,294,967,295 inclusive. All 32-bit integer values are represented in *little endian* format with the least significant word and least significant byte at the lowest address. `Long` and `unsigned long` occupy 32 bits as this is the smallest long integer size allowed by the ANSI standard for C.

Variables may be declared using the `signed long int` and `unsigned long int` keyword sequences, respectively, to hold values of these types. Where only `long int` is used in the declaration, the type will be `signed long`. When specifying this type, the keyword `int` may be omitted. Thus a variable declared as `long` will contain a `signed long int` and a variable declared as `unsigned long` will contain an `unsigned long int`.

Table 3.3: Floating-point formats

Format	Sign	biased exponent	mantissa
IEEE 754 32-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
modified IEEE 754 24-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx

Table 3.4: Floating-point format example IEEE 754

Format	Number	biased exponent	1.mantissa	decimal
32-bit	7DA6B69Bh	11111011b	1.01001101011011010011011b	2.77000e+37
		(251)	(1.302447676659)	
24-bit	42123Ah	10000100b	1.001001000111010b	36.557
		(132)	(1.142395019531)	

3.3.8 Floating Point Types and Variables

Floating point is implemented using either a IEEE 754 32-bit format or a modified (truncated) 24-bit form of this.

The 24-bit format is the default for all `float` and `double` values. This can be explicitly set using the `--float=24` or `--double=24` option. The 32-bit format is used for `double` values if the `--double=32` option is used. All `float` values can be set to 32 bits wide by using the `--float=32` option or `--float=double` if the `double` type is also set to 32 bits wide.

This format is described in 3.3, where:

- sign is the sign bit
- The exponent is 8-bits which is stored as *excess 127* (i.e. an exponent of 0 is stored as 127).
- mantissa is the mantissa, which is to the right of the radix point. There is an implied bit to the left of the radix point which is always 1 except for a zero value, where the implied bit is zero. A zero value is indicated by a zero exponent.

The value of this number is $(-1)^{sign}$ x $2^{(exponent-127)}$ x 1.mantissa.

Here are some examples of the IEEE 754 32-bit formats:

Note that the most significant bit of the mantissa column in 3.4 (that is the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero (in which case the float is zero).

The 32-bit example in 3.4 can be calculated manually as follows.

The sign bit is zero; the biased exponent is 251, so the exponent is 251-127=124. Take the binary number to the right of the decimal point in the mantissa. Convert this to decimal and divide it by 2^{23}

where 23 is the number of bits taken up by the mantissa, to give 0.302447676659. Add one to this fraction. The floating-point number is then given by:

$-1^0 \times 2^{124} \times 1.302447676659 = 1 \times 2.126764793256e+37 \times 1.302447676659 \approx 2.77000e+37$

Variables may be declared using the `float` and `double` keywords, respectively, to hold values of these types. Floating point types are always signed and the `unsigned` keyword is illegal when specifying a floating point type. Types declared as `long double` will use the same format as types declared as `double`.

3.3.9 Structures and Unions

HI-TECH C Compiler for PIC18 MCUs supports `struct` and `union` types of any size from one byte upwards. Structures and unions only differ in the memory offset applied for each member. The members of structures and unions may not be objects of type `bit`, but bit-fields are fully supported.

Structures and unions may be passed freely as function arguments and return values. Pointers to structures and unions are fully supported.

3.3.9.1 Bit-fields in Structures

HI-TECH C Compiler for PIC18 MCUs fully supports *bit-fields* in structures.

Bit-fields are allocated within 8- or 16-bit words. Although the ANSI standard only allows for bit-fields of type `int` or `unsigned int`, this is not optimial on PICC18 devices. The allocation size of the bit-field structure is based on the number of bits defined in the structure as a whole, thus the following structure:

```
struct {
        unsigned        lo : 1;
        unsigned        dummy : 6;
        unsigned        hi : 1;
} foo;
```

will be allocated 1 byte of memory in total, but the following:

```
struct {
        unsigned        lo : 1;
        unsigned        dummy : 6;
        unsigned        hi : 1;
        unsigned        extra : 2;
} foo;
```

will be allocated 2 bytes of storage.

Unnamed bit-fields may be declared to pad out unused space between active bits in control registers. For example, if dummy is never used the structure above could have been declared as:

```
struct {
        unsigned        lo : 1;
        unsigned           : 6;
        unsigned        hi : 1;
} foo;
```

A structure with bit-fields may be initialised by supplying a comma-separated list of initial values for each field. For example:

```
struct {
        unsigned        lo  : 1;
        unsigned        mid : 6;
        unsigned        hi  : 1;
} foo = {1, 8, 0};
```

As PIC18 devices are little endian, the first bit defined will be the least significant bit of the word in which it will be stored. When a bit-field is declared, it is allocated within the current word if it will fit, otherwise a new word is allocated within the structure. Bit-fields can never cross the boundary between word allocation units. For example, the declaration:

```
struct {
        unsigned        lo : 1;
        unsigned        dummy : 6;
        unsigned        hi : 1;
} foo;
```

will produce a structure occupying 1 byte. If foo was ultimately linked at address 10H, the field lo will be bit 0 of address 10H, hi will be at bit 7. The least significant bit of dummy will be bit 1 of address 10H and the most significant bit of dummy will be at bit 6.

3.3.9.2 Structure and Union Qualifiers

HI-TECH C Compiler for PIC18 MCUs supports the use of type qualifiers on structures. When a qualifier is applied to a structure, all of its members will inherit this qualification. In the following example the structure is qualified const.

```
const struct {
        int number;
        int *ptr;
} record = { 0x55, &i};
```

In this case, the structure will be placed into the program space and each member will, obviously, be read-only. Remember that all members must be initialized if a structure is `const` as they cannot be initialized at runtime.

If the members of the structure were individually qualified `const` but the structure was not, then the structure would be positioned into RAM, but each member would be read-only. Compare the following structure with the above.

```
struct {
        const int number;
        int * const ptr;
} record = { 0x55, &i};
```

3.3.10 Standard Type Qualifiers

Type qualifiers provide information regarding how an object may be used, in addition to its type which defines it storage size and format. HI-TECH C Compiler for PIC18 MCUs supports both ANSI qualifiers and additional special qualifiers which are useful for embedded applications and which take advantage of the PIC18 architecture.

3.3.10.1 Const and Volatile Type Qualifiers

HI-TECH C Compiler for PIC18 MCUs supports the use of the ANSI type qualifiers `const` and `volatile`.

The `const` type qualifier is used to tell the compiler that an object is read only and will not be modified. If any attempt is made to modify an object declared `const`, the compiler will issue a warning. User-defined objects declared `const` are placed in a special psects in the program space. Obviously, a `const` object must be initialised when it is declared as it cannot be assigned a value at any point at runtime. For example:

```
const int  version = 3;
```

The `volatile` type qualifier is used to tell the compiler that an object cannot be guaranteed to retain its value between successive accesses. This prevents the optimizer from eliminating apparently redundant references to objects declared `volatile` because it may alter the behaviour of the program to do so. All Input/Output ports and any variables which may be modified by interrupt routines should be declared `volatile`, for example:

```
volatile static near unsigned char  PORTA @ 0xF80;
```

Volatile objects may be accessed using different generated code to non-volatile objects. For example, when assigning a non-volatile object the value 1, the object may be cleared and then incremented, but the same operation performed on a volatile object will load the W register with 1 and then store this to the appropriate address.

3.3.11 Special Type Qualifiers

HI-TECH C Compiler for PIC18 MCUs supports special type qualifiers, persistent, near and far to allow the user to control placement of static and extern class variables into particular address spaces. If the PICC18 option, --STRICT is used, these type qualifiers are changed to __persistent, __near and __far, respectively. These type qualifiers may also be applied to pointers. These type qualifiers may not be used on variables of class auto; if used on variables local to a function they must be combined with the static keyword. For example, you may not write:

```
void test(void) {
   persistent int intvar; /* WRONG! */
   ... other code ...
}
```

because intvar is of class auto. To declare intvar as a persistent variable local to function test(), write:

```
static persistent int intvar;
```

HI-TECH C Compiler for PIC18 MCUs also supports the keywords bank1, bank2 and bank3. These keywords have been included to allow code to be easily ported from HI-TECH C Compiler for PIC10/12/16 MCUs. These keywords are accepted by HI-TECH C Compiler for PIC18 MCUs, but have no effect in terms of the object's storage or how they are accessed. The --ADDRQUAL option has no effect on these qualifiers in this version of the compiler.

3.3.11.1 Persistent Type Qualifier

By default, any C variables that are not explicitly initialised are cleared to zero on startup. This is consistent with the definition of the C language. However, there are occasions where it is desired for some data to be preserved across resets or even power cycles (on-off-on).

The persistent type qualifier is used to qualify variables that should not be cleared on startup. In addition, any persistent variables will be stored in a different area of memory to other variables. Persistent objects are placed within one of the non-volatile psects. If the persistent object is also qualified near, it placed in the nvrram psect. Persistent bit objects are placed within the nvbit psect. All other persistent objects are placed in the nvram psect.

3.3.11.2 Near Type Qualifier

The `near` type qualifier can be used to place non-`auto` variables in the *access bank* of the PIC18. The access bank is always accessible, regardless of the currently selected RAM bank, so accessing `near` objects may be faster than accessing other objects and typically results in smaller code sizes.

The compiler automatically uses the access bank for frequently accessed user-defined variables so this qualifier would only be needed for special memory placement of objects, for example if C variables are accessed in hand-written assembly code that assumes that they are located in this memory.

This qualifier is controlled by the compiler option `--ADDRQUAL`, which determines its effect, see Section 2.6.18. Based on this option's settings, this qualifier may be binding or ignored (which is the default operation). Qualifiers which are ignored will not produce an error or warning, but will have no effect.

Here is an example of an `unsigned char` object placed within the access bank:

```
near unsigned char fred;
```

Objects qualified `near` cannot be `auto` or parameters to a function, but can be qualified `static`, allowing them to be defined locally within a function, as in:

```
void myFunc(void) {
static near unsigned char local_near;
```

Note that the compiler may store some temporary objects in the common memory, so not all of this space may be available for user-defined variables.

If the PICC18 option, `--STRICT` is used, this type qualifier is changed to `__near`.

For the operation of this qualifier in C18 compatibility mode (see Section 3.1.4), refer to the MPLAB C Compiler for PIC18 MCUs manual.

3.3.11.3 Far Type Qualifier

The `far` type qualifier is used to place non-auto variables into the program memory space for those PIC18 devices which can support external memory. The compiler assumes that variables will be located in RAM in this memory space.

Accesses to `far` variables are less efficient than that to internal variables and will result in larger, slower code.

This qualifier is controlled by the compiler option `--ADDRQUAL`, which determines its effect, see Section 2.6.18. Based on this option's settings, this qualifier may be binding or ignored (which is the default operation). Qualifiers which are ignored will not produce an error or warning, but will have no effect.

Here is an example of an `unsigned int` object placed into the device's external code space:

```
far unsigned int farvar;
```

Objects qualified `far` cannot be `auto` or parameters to a function, but can be qualified `static`, allowing them to be defined locally within a function, as in:

```
void myFunc(void) {
static far unsigned char local_far;
```

If the PICC18 option, `--STRICT` is used, this type qualifier is changed to `__far`.

Note that not all devices support extending their memory space in this way and the `far` qualifier is not applicable to all PIC18 devices. For those devices that can extend their memory, the address range where the additional memory will be mapped must first be specified with the `--RAM` option, see 2.6.53. For example, to map additional data memory from 20000h to 2FFFFh use `--RAM=default,+20000-2FFFF`.

For the operation of this qualifier in C18 compatibility mode (see Section 3.1.4), refer to the MPLAB C Compiler for PIC18 MCUs manual.

3.3.12 Pointer Types

There are two basic pointer types supported by HI-TECH C Compiler for PIC18 MCUs: data pointers and function pointers. Data pointers hold the address of variables which can be read, and possible written, indirectly by the program. Function pointers hold the address of an executable routine which can be called indirectly via the pointer.

Typically qualifiers are used with pointer definitions to customise the scope of the pointer, allowing the code generator to set an appropriate size and format for the addresses the pointer will hold. PRO version compilers use sophisticated algorithms to track the assignment of addresses to data pointers, and, as a result, many of these qualifiers no longer need to be used, and the size of the pointer is optimal for the its intended usage.

It is helpful to first review the ANSI standard conventions for definitions of pointer types.

3.3.12.1 Combining Type Qualifiers and Pointers

Pointers can be qualified like any other C object, but care must be taken when doing so as there are two quantities associated with pointers. The first is the actual *pointer* itself, which is treated like any ordinary C variable and has memory reserved for it. The second is the *target* that the pointer references, or to which the pointer points. The general form of a pointer definition looks like the following.

```
target_type_&_qualifiers * pointer's_qualifiers pointer's_name;
```

Any qualifiers to the right of the * (i.e. next to the pointer's name) relate to the pointer variable itself. The type and any qualifiers to the left of the * relate to the pointer's targets.

TUTORIAL

EXAMPLE OF POINTER QUALIFIERS Here are three examples of pointer definitions using the volatile qualifier. The fields in the definitions have been highlighted with spacing:

```
volatile int *           vip ;

int          * volatile ivp ;

volatile int * volatile vivp ;
```

The first example is a pointer called `vip`. It contains the address of `int` objects that are qualified `volatile`. The pointer itself — the variable that holds the address — is *not* `volatile`, however the objects that are accessed when the pointer is dereferenced are `volatile`. That is, the target objects accessible via the pointer may be externally modified.

The second example is a pointer called `ivp` which also contains the address of `int` objects. In this example, the pointer itself is `volatile`, that is, the address the pointer contains may be externally modified, however the objects that can be accessed when dereferencing the pointer are not `volatile`.

The last example is of a pointer called `vivp` which is itself qualified `volatile` and which also holds the address of a `volatile` object.

Bear in mind that one pointer can be assigned the address of many objects, for example a pointer that is a parameter to a function is assigned a new object address every time the function is called. The definition of the pointer must be valid for every target address assigned.

•

Care must be taken when describing pointers: Is a "const pointer" a pointer that points to `const` objects, or a pointer that is `const` itself. You can talk about "pointers to const" and "const pointers" to help clarify the definition, but such terms may not be universally understood.

3.3.12.2 Data Pointers

HI-TECH C Compiler for PIC18 MCUs monitors and records all assignments of addresses to each data pointer the program defines. The size and format of the address held by each pointer is based on this information. When more than one address is assigned to a pointer at different places in the code, a set of all possible targets the pointer can address is maintained. This information is specific to each pointer defined in the program, thus two pointers with the same type may hold addresses of different sizes and formats due to the different nature of objects they address in the program.

The following pointer classifications are currently implemented:

- An 8-bit pointer capable of accessing the access bank;
  - Address is an offset into the access bank
- A 16-bit pointer capable of accessing the entire data memory space;
- An 8-bit pointer capable of accessing up to 256 bytes of program space data;
  - Address is an offset into psect `smallconst`;
- A 16-bit pointer capable of accessing up to 64 kbytes of program space data;
  - Address is an offset into psect `mediumconst` which is linked into any 64k block, but with an offset into this block equal to the size of the data space memory;
- A 24-bit pointer capable of accessing the entire program space;
- A 16-bit pointer capable of accessing the entire data space memory and up to 64 kbytes of program space data;
  - Addresses above the top of the data space access program space; other addresses access data space;
- A 24-bit pointer capable of accessing the entire data space memory and the entire program space;
  - Bit #21 determines destination: this bit set indicates a data space address; clear indicates a program space address
  - This is the default pointer configuration as it can point to any object.

Each data pointer will be allocated one of the above classifications after preliminary scans of the source code. There is no mechanism by which the programmer can specify the style of pointer required (other than by the address assignments to the pointer).

TUT●RIAL

DYNAMIC POINTER SIZES A program in the early stages of development contains the following code;

```
void main(void) {
int i, *ip;
ip = &i;
}
```

The code generator is able to automatically allocate the variable `i` to the access bank, which it does. The code generator notes that the pointer `ip` only points to the access bank variable `i`, so this pointer is made an 8-bit wide access bank pointer.

As the program is developed, other `near` variables are defined and allocated space in the access bank. A point is reach at which the variable `i` will no longer fit in the access bank and it is automatically moved to banked RAM. When the program is next compiled, the pointer `ip` will automatically become a 16-bit pointer to all of the data space, and the code used to initialize and dereference the pointer will change accordingly.

One positive aspect of tracking pointer targets is less of a dependence on pointer qualifiers. The standard qualifiers `const` and `volatile` must still be used in pointer definitions to indicate a read-only or externally-modifiable target object, respectively. However this is in strict accordance with the ANSI standard. HI-TECH specific qualifiers, like `near` and `far`, do not need to be used to indicate pointer targets, and should be avoided. The non-use of these qualifiers will result in more portable and readable code, and lessen the chance of extraneous warnings being issued by the compiler.

3.3.12.3 Pointers to Const

The `const` qualifier plays no direct part in specifying the pointer classification that the compiler will allocate to a pointer. This qualifier should be used when the target, or targets, referenced by the pointer should be read-only. The addresses of `const` objects assigned to a pointer will result in that pointer having a classification capable of accessing the program space. The exact classification will also depend on other factors.

The code generator tracks the total size of `const` qualified variables that are defined. It uses this information to determine how large any pointers that can access `const` objects must be. Such pointers may be either 1, 2 or 3 bytes wide.

TUT●RIAL

POINTERS AND CONST DATA Assume a program contains of the following:

```
void main(void) {
const char in_table[20] = { /* values */ };
char * cp;
cp = &in_table;
}
```

If the array above is the only `const` data in the program, then there are 20 bytes of `const` data used in the program. In this instance, the code generator will make the pointer, `cp`, a one byte wide pointer to objects in the program space.

Later, the program is changed and another `const` array is added to the code:

```
const char out_table[200] = { /* values */ };
```

As the total size of `const` data for this program now exceeds 255 bytes, the size of *any* pointer that can access `const` objects will be made 2 bytes long. Even if the pointer, `cp`, is not assigned the address of this new array, `out_table`, its size will increase.

For pointer that are accessing const objects, the address contained within the pointer is an offset into the psect used to store the const data. For programs defining less than 256 bytes of const data, this data is placed into a psect called `smallconst`; for larger const data amounts up to 64 kbytes, the psect is called `mediumconst`.

The size of pointers that can access const data indicates the storage size of the address. However, PIC18 devices use a 3-byte table pointer SFR to access data in the program space and all 3 bytes of this register must be loaded and valid to access program space data. To avoid having to load all 3 bytes of this register with each program space access, the code generator also keeps track of the number of table pointer registers that are modified during the program. These active table pointer registers will be re-loaded with each program space access. The non-active registers are assumed to retain the value assigned to them in the runtime startup code.

Any hand-written assembler code, or C code that writes to the table pointer SFRs directly, must ensure that the contents of any non-active table registers are preserved. Saving both TBLPTRH and TBLPTRU will ensure that this requirement is met.

3.3.12.4 Pointers to Both Memory Spaces

When a pointer is assigned the address of one or more objects allocated memory in the data space, and also assigned the address of one or more `const` objects, the pointer will be classified such that

it can dereference both memory spaces, and the address will be encoded so that the target memory space can be determined at runtime.

A 16-bit mixed space pointer is encoded such that if it holds an address that is higher than the highest general purpose RAM address, it holds the address of a program space object; all other address reference objects in the data space.

A 24-bit mixed space pointer is encoded such that if bit number #21 is set, it contains the address of an object in the data space; all other addresses hold the address of a program space object.

TUTORIAL

POINTERS TO DIFFERENT TARGETS A program in the early stages of development contains the following code;

```
int getValue(int * ip) {
return 2 + *ip ;
}
void main(void) {
int j, i = setV();
j = getValue(&i)
}
```

The code generator allocate the variable `i` to the access bank and the pointer `ip` (the parameter to the function `getValue`) is made an 8-bit wide access bank pointer. At a later date, the function `main` is changed, becoming:

```
void main(void) {
int j, i = setV();
const int start = 0x10;
j = getValue(&i)
j += getValue(&start);
}
```

Now the pointer, `ip`, is assigned addresses of both data and `const` objects. After the next compilation the size and encoding of `ip` will change, as will the code that assigns the addresses to `ip`. The generated code that dereferences `ip` (in `getValue`) will check the address to determine the memory space of the target address.

3.3.12.5 Function Pointers

Function pointers can be defined to indirectly call functions or routines in the program space. The size of these pointers are 16 or 24 bits wide and is determined by the amount of program memory

defined. Function pointers are 16-bits wide for memory spaces less the 64 kbytes in size. For larger program space memory sizes, these then swap to 3 bytes in size.

It should be stressed that direct calls to functions are not affected by the size of function pointers. The size of function pointers only affect code calling functions indirectly. The addresses for all code labels are always shown in the map file as an untruncated 3-byte address regardless of the pointer size determined by the code generator.

The size of function pointers will affect the number of table pointer registers considered active.

3.4 Storage Class and Object Placement

Objects are positioned in different memory areas dependant on their storage class and declaration. This is discussed in the following sections.

3.4.1 Local Variables

A *local variable* is one which only has scope within the block in which it was defined. That is, it may only be referenced within that block. C supports two classes of local variables in functions: `auto` variables which are normally allocated in a compiled stack, and `static` variables which are always given a fixed memory location and have permanent duration.

3.4.1.1 Auto Variables

This section discusses allocation of `auto` variables (those with automatic storage duration). This also include function parameter variables, which behave like `auto` variables, as well as temporary variables defined by the compiler.

The `auto` (short for automatic) variables are the default type of local variable. Unless explicitly declared to be `static`, a local variable will be made `auto`. The `auto` keyword may be used if desired.

The `auto` variables, as their name suggests, automatically come into existence when a function is executed, then disappear once the function returns. Since they are not in existence for the entire duration of the program, there is the possibility to reclaim memory they use when the variables are not in existence and allocate it to other variables in the program.

Typically such variables are stored on some sort of a data stack, which can easily allocate then deallocate memory as required by each function. All devices targeted by the compiler do not have a data stack that can easily be operated in this fashion. As a result, an alternative stack construct is implemented by the compiler. The stack mechanism employed is known as a compiled stack and is fully described in Section 3.4.1.1.

Once `auto` variables have been allocated a relative position in the compiled stack, the stack itself is then allocated memory in the data space. This is done is a similar fashion to the way non-`auto`

variables are assigned memory: a psect is used to hold the stack and this psect is placed into the available data memory by the linker. The psect used to hold the compiled stack is called `cstack`, and like with non-auto variable psects, the psect basename is always used in conjunction with a linker class name to indicate the RAM bank in which the psect will be positioned. See Section 3.8.1 for the limitations associated with where this psect can be linked.

The `auto` variables defined in a function will not necessarily be allocated memory in the order declared, in contrast to parameters which are always allocated memory based on their lexical order. In fact, `auto` variables for one function may be allocated in many RAM banks.

The the standard qualifiers: `const` and `volatile` may both be used with `auto` variables and these do not affect how they are positioned in memory. This implies that a local `const`-qualified object is still an `auto` object and, as such, will be allocated memory in the compiled stack in the data space memory, not in the program memory like with non-`auto` `const` objects.

The compiler will try to locate the stack in one data bank, but if this fills (i.e. if the compiler detects that the stack has become too large for the free space in a bank), it can build up the stack into several components (each with their own psect) and link each in a different bank.

Each `auto` object is referenced in assembly code using a special symbol defined by the code generator. If you write assembly code that accesses `auto` variables defined in C source code, you must use these symbols, which are discussed in Section 3.10.3.

Compiled Stack Operation A compiled stack consists of fixed memory areas that are usable by each function's stack-based variables. When a compiled stack is used, functions are not re-entrant since stack-based variables in each function will use the same fixed area of memory every time the function is invoked.

Fundamental to the generation of the compiled stack is the call graph, which defines a tree-like hierarchy of function calls, i.e it shows what functions may be called by each function.

There will be one graph produced for each root function. A root function is typically not called, but which is executed via other means and contains a program entry point. The function `main()` is an example of a root function that will be in every project. Interrupt functions, which are executed when a hardware interrupt occurs, are another example.

Figure 3.1 shows sections of a program being analyzed by the code generator to form a call graph. In the original source code, the function main() calls `F1()`, `F2()` and `F3()`. `F1()` calls `F4()`, but the other two functions make no calls. The call graph for `main()` indicates these calls. The symbols `F1`, `F2` and `F3` are all indented one level under main. `F4` is indented one level under `F1`.

This is a static call graph which shows all possible calls. If the exact code for function `F1()` looked like:

```
int F1(void) {
   if(PORTA == 44)
      return F4();
```

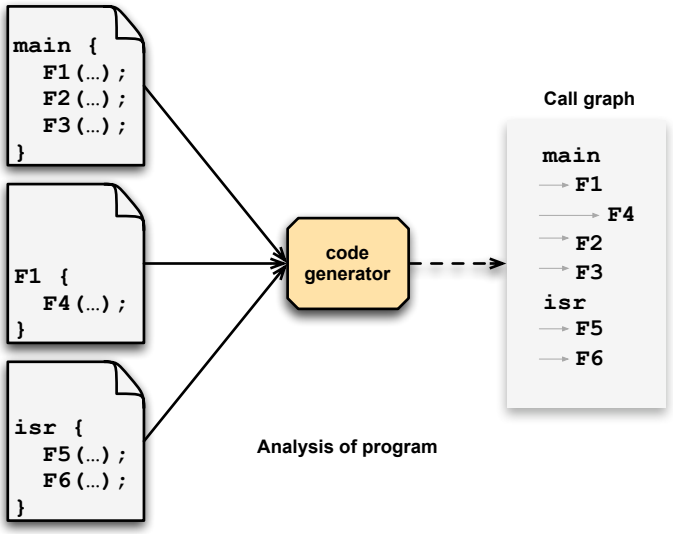


Figure 3.1: Formation of Call Graph

```
    return 55;
}
```

the function `F4()` will always appear in the call graph, even though it is conditionally executed in the actual source code. Thus, the call graph indicates all functions, even those that might be called.

In the diagram, there is also an interrupt function, `isr()`, and it too has a separate graph generated.

The term main-line code is often used, and refers to any code that is executed as a result of the `main()` function being executed. In the above figure, `F1()`, `F2()`, `F3()` and `F4()` are only ever called by main-line code.

The term interrupt code refers to any code that is executed as a result of an interrupt being generated, in the above figure, `F5()` and `F6()` are called by interrupt code.

Figure 3.2 graphically shows an example of how the compiled stack is formed.

Each function in the program is allocated a block of memory for its parameter, auto and temporary variables. Each block is referred to as an auto-parameter block (APB). The figure shows the APB being formed for function `F2()`, which has two parameters, `a` and `b`, and one `auto` variable, `c`.

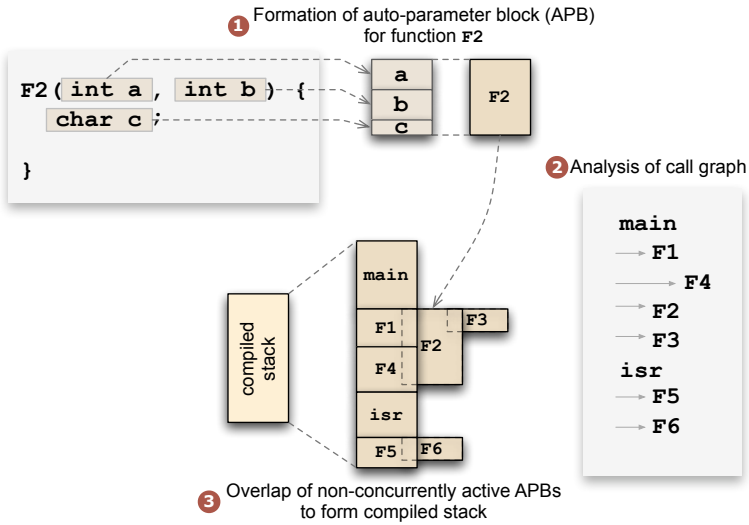


Figure 3.2: Formation of the Compiled Stack

The parameters to the function are first grouped in an order strictly determined by the lexical order in which they appear in the source code. These are then followed by any `auto` objects, however the `auto` objects may be placed in any order. So we see memory for a is followed by that for `b` and lastly, `c`.

Once these variables have been grouped, the exact location of each object is not important at this point and we can represent this memory by one block — the APB for this function.

The APBs are formed for all functions in the program. Then, by analyzing the call graph, these blocks are assigned positions, or bases values, in the compiled stack.

Memory can be saved if the following point is observed: If two functions are never active at the same time, then their APBs can be overlapped.

In the example shown in the figure, `F4()` and `F1()` are active at the same time, in fact `F1()` calls `F4()`. However `F2()`, `F3()` and `F1()` are never active at the same time; `F1()` must return before `F2()` or `F3()` can be called by `main()`. The function `main()` will always be active and so its APB can never overlap with that of an other function.

In the compiled stack, you can see that the APB for `main()` is allocated unique memory. The blocks for `F1()`, `F2()` and `F3()` are all placed on top of each other and the same base value in the compiled stack, however the memory taken up by the APBs for `F1()` and `F4()` are unique and do not overlap.

Our example also has an interrupt function, `isr()`, and its call graph is used to assemble the

APBs for any interrupt code in the same way. Being the root of a graph, `isr()` will always be allocated unique memory, and the APBs for interrupt functions will be allocated memory following.

The end result is a block of memory which forms the compiled stack. This block can then be placed into the device's memory by the linker.

For devices with more than one bank of data memory, the compiled stack may be built up into components, each located in a different memory bank. The compiler will try to allocate the compiled stack in one bank, but if this fills, it will consider other banks. The process of building these components of the stack is the same, but each function may have more than one APB and these will be allocated to one of the stack components based on the remaining memory in the component's destination bank.

Human readable symbols are defined by the code generator which can be used to access `auto` and parameter variables in the compiled stack from assembly code, if required. See Section 3.10.4 for full information between C domain and assembly domain symbols.

3.4.1.2 Static Variables

All `static` variables have permanent storage duration, even those defined inside a function which are "local static" variables. Local static variables only have scope in the function or block in which they are defined, but unlike `auto` variables, their memory is reserved for the entire duration of the program. Thus they are allocated memory like other non-`auto` variables. Any `static` variable may be accessed by other functions via pointers since they have permanent duration.

Variables which are `static` are guaranteed to retain their value between calls to a function, unless explicitly modified via a pointer.

Variables which are `static` and which are initialized only have their initial value assigned once during the program's execution. Thus, they may be preferable over initialized `auto` objects which are assigned a value every time the block in they are defined begins execution. Any initialized `static` variables are initialized in the same way as other non-`auto` initialized objects by the runtime startup code, see Section 2.3.2.

Local objects which are `static` are assigned an assembly symbol which consists of the function name followed by an `@` symbol and the variable's lexical name, e.g. `main@foobar` will be the assembly identifier used for the `static` variable `foobar` defined in `main()`.

Non-local `static` objects use their lexical name with a leading underscore character, e.g. `_foobar` will be the assembly identifier used for this object. However, if there is more than one such `static` object defined, then subsequent objects will use the name of the file that contains them and their lexical name separated by an `@` symbol, e.g. `lcd@foobar` would be the assembly symbol for the `static` variable `foobar` defined in `lcd.c`.

3.4.2 Absolute Variables

Most variables can be located at an absolute address by following its declaration with the construct `@ address`, where `address` is the location in memory where the variable is to be positioned. Such a variables is known as an absolute variables.

Defining absolute objects can fragment memory and may make it impossible for the linker to position other objects. Avoid absolute objects if at all possible. If absolute objects must be defined, try to place them at either end of a memory bank or page so that the remaining free memory is not frag- mented into smaller chunks.

3.4.2.1 Absolute Variables in Data Memory

Absolute variables are primarily intended for equating the address of a C identifier with a special function register, but can be used to place ordinary variables at an absolute address in data memory.

For example:

```
volatile unsigned char Portvar @ 0x06;
```

will declare a variable called `Portvar` located at 06h in the data memory. The compiler will reserve storage for this object and will equate the variable's identifier to that address. The compiler-generated assembler will include a line similar to:

```
_Portvar EQU 06h
```

No `auto` variables can be made absolute as they are located in a compiled stack. See Section 3.4.1.1. Absolute variables cannot be initialized.

The compiler does not make any checks for overlap of absolute variables with other absolute variables, so this must be considered when choosing the variable locations. There is no harm in defining more than one absolute variable to live at the same address if this is what you require. The compiler will not locate ordinary variables over the top of absolutes, so there is no overlap between these objects.

3.4.2.2 Absolute Variables in Program Memory

Non-`auto` objects qualified `const` can also be made absolute in the same way, however the address will indicate an address in program memory. For example:

```
const int settings[] @ 0x200 = { 1, 5, 10, 50, 100 };
```

Both initialized and uninitialized `const` objects can be made absolute. That latter is useful when you only need to define a label in program memory without making a contribution to the output file.

3.4.3 Objects in Program Space

`Const` objects are usually placed in program space. On the PIC18 devices, the program space is byte-wide, the compiler stores one character per byte location and values are read using the table read instructions. All `const`-qualified data objects and string literals are placed in either the `smallconst`, `mediumconst` or `const` psect, depending on the amount of const data defined in the program. The appropriate const psect is placed at an address above the upper limit of RAM since RAM and `const` pointers use this address to determine if an access to ROM or RAM is required. See Section 3.3.12.

3.4.4 Dynamic Memory Allocation

Dynamic memory allocation, (heap-based allocation using `malloc` etc.) is not supported with HI-TECH C. This is due to the limited amount of data memory and the fact that this memory is banked. The wasteful nature of dynamic memory allocation does not suit itself to the 8-bit PIC18 device architectures.

3.4.5 Memory Models

HI-TECH C does not use fixed memory models to alter allocation of variables to memory. Memory allocation is fully automatic and there are no memory model controls.

3.5 Functions

In some situations, the code associated with a function is output more than once. See Section 3.9.4 from more information.

3.5.1 Absolute Functions

The generated code associated with a function can be placed at an absolute address. This can be accomplished by using an `@ address` construct in a similar fashion to that used with absolute variables.

The following example of an absolute function which will place the function label and first assembly instruction corresponding to the function at address 400h:

```
int mach_status(int mode) @ 0x400
{
    /* function body */
}
```

Using this construct with interrupt functions will not alter the position of the interrupt context saving code that precedes the code associated with the interrupt function body. See also Section 2.6.24.

3.5.2 External Functions

If a call to a function that is defined outside the program C source code is required (it may be part of code compiled separately, e.g. bootloader, or assembly code), you will need to provide a declaration of the function so that the compiler knows how to encode the call.

If this function takes arguments or returns a value, the compiler may use a symbol to represent the memory locations used to store these values, see Sections 3.5.3 and 3.5.4 to determine if a register or memory locations are used in this transfer. If an argument or return value is used and this will be stored in memory, the corresponding symbol must be defined by your code and assigned the value of the appropriate memory location.

The value can be determined from the map file of the external build, which compiled the function, or from the assembly code. If the function was written in C, look for the symbol `?_funcName`, where *funcName* is the name of the function. It can be defined in the program which makes the call via a simple `EQU` directive in assembler. For example, the following could be placed in the C source.

```
#asm
    GLOBAL ?_extReadFn
    ?_extReadFn EQU 0x20
#endasm
```

Alternatively, the assembly code could be contained directly in an assembly module.

If this symbol is not defined, the compiler will issue an undefined symbol error. This error can be used to verify the name being used by the compiler to encode the call, if required.

It is not recommended to call the function indirectly by casting an integer to a function pointer, but in such a circumstance, the compiler will use the value of the constant in the symbol name, for example calling a function at address 200h will require the definition of the symbol `?0x200` to be the location of the parameter/return value location for the function.

Note that the return value of a function (if used) shares the same locations assigned to any parameters to that function and both use the same symbol.

3.5.3 Function Argument Passing

HI-TECH C uses a fixed convention to pass arguments to a function. The method used to pass the arguments depends on the size and number of arguments involved.

The names “argument” and “parameter” are often used interchangeably, but typically an argument is the actual value that is passed to the function and a parameter is the variable defined by the function to store the argument.

The compiler will either pass arguments in the W register, or in the auto-parameter block (APB) of the called function. If the first parameter is one byte in size, it is passed in the W register. All other parameters are passed in the APB. This applies to basic types and to aggregate types, like structures.

The parameters are grouped along with the function’s `auto` variables in the APB and are placed in the compiled stack. See Section 3.4.1.1 for detailed information on the compiled stack. The parameter variables will be referenced as an offset from the symbol `?_function`, where function is the name of the function in which the parameter is defined (i.e. the function that is to be called).

Unlike `auto` variables, parameter variables are allocated memory strictly in the order in which they appear in the function’s prototype. This means that the parameters will always be placed in the same memory bank even if the other `auto` variables for that function have been allocated across multiple banks.

The parameters for functions that take a variable argument list (defined using an ellipsis in the prototype) are placed in the parameter memory, along with named parameters.

Take, for example, the following ANSI-style function.

```
void test(char a, int b);
```

The function `test()` will receive the argument for `b` in its function auto-parameter block and that for `a` in the W register. A call to this function:

```
test(xyz, 8);
```

would generate code similar to:

```
MOVLW 08h      ; move literal 0x8 into...
MOVWF ?_test   ; the auto-parameter memory...
CLRF ?_test+1  ; locations for the 16-bit parameter
MOVF _xyz,w    ; move xyz into the W register
CALL (_test)
```

In this example, the parameter `b` is held in the memory locations `?_test` (Least Significant Byte) and `?_test+1` (Most Significant Byte) which are on the compiled stack. You may also see this same location referenced as `test@b`, which is an alternate symbol.

The exact code used to call a function, or the code used to access a parameters from within a function, can always be examined in the assembly list file. See Section 2.6.19 for the option that generates this file. This is useful if you are writing an assembly routine that must call a function with parameters, or accept arguments when it is called. The above example does not consider data memory banking or program memory paging, which may require additional instructions.

3.5.4 Function Return Values

Function return values are passed to the calling function using either the W register, or the function's auto-parameter block. Having return values also located in the same memory as that used by the parameters can reduce the code size for functions that return a modified copy of their parameter.

Eight-bit values are returned from a function in the W register. Values larger than a byte are returned in the function's parameter memory area, with the least significant word in the lowest memory location. This memory area is a block that can be accessed with an offset from the symbol `?_`*`funcName`*, where *`funcName`* is the name of the function that returns the value.

For example, the function:

```
int return_16(void){
   return 0x1234;
}
```

will exit with the code similar to the following :

```
MOVLW   34h
MOVWF   (?_return_16)
MOVLW   12h
MOVWF   (?_return_16)+1
RETURN
```

3.5.4.1 Structure Return Values

Return values that have an aggregate type (e.g. `struct` and `union` types), but whose size is 4 bytes or smaller, are returned in the parameter memory, as is done with return values of basic type. For aggregate return values greater than 4 bytes in size, the object is also copied to the base of the function's parameter area and the address of the copy is returned in the `FSR0` register.

3.6 Operators

HI-TECH C Compiler for PIC18 MCUs supports all the ANSI operators. The exact results of some of these are implementation defined. The following sections illustrate code produced by the compiler.

3.6.1 Integral Promotion

When there is more than one operand to an operator, they typically must be of exactly the same type. The compiler will automatically convert the operands, if necessary, so they have the same

type. The conversion is to a "larger" type so there is no loss of information. Even if the operands have the same type, in some situations they are converted to a different type before the operation. This conversion is called *integral promotion*. HI-TECH C Compiler for PIC18 MCUs performs these integral promotions where required. If you are not aware that these changes of type have taken place, the results of some expressions are not what would normally be expected.

Integral promotion is the implicit conversion of enumerated types, `signed` or `unsigned` varieties of `char`, `short int` or bitfield types to either `signed int` or `unsigned int`. If the result of the conversion can be represented by an `signed int`, then that is the destination type, otherwise the conversion is to `unsigned int`.

Consider the following example.

```
unsigned char count, a=0, b=50;
if(a - b < 10)
   count++;
```

The `unsigned char` result of `a - b` is 206 (which is not less than 10), but both `a` and `b` are converted to `signed int` via integral promotion before the subtraction takes place. The result of the subtraction with these data types is -50 (which is less than 10) and hence the body of the `if()` statement is executed. If the result of the subtraction is to be an `unsigned` quantity, then apply a cast. For example:

```
if((unsigned int)(a - b) < 10)
 count++;
```

The comparison is then done using unsigned int, in this case, and the body of the if() would not be executed.

Another problem that frequently occurs is with the bitwise compliment operator, "~". This operator toggles each bit within a value. Consider the following code.

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
   count++;
```

If `c` contains the value 55h, it often assumed that ~c will produce AAh, however the result is FFAAh and so the comparison above would fail. The compiler may be able to issue a mismatched comparison error to this effect in some circumstances. Again, a cast could be used to change this behaviour.

The consequence of integral promotion as illustrated above is that operations are not performed with `char`-type operands, but with `int`-type operands. However there are circumstances when the result of an operation is identical regardless of whether the operands are of type `char` or `int`. In these cases, HI-TECH C Compiler for PIC18 MCUs will not perform the integral promotion so as to increase the code efficiency. Consider the following example.

Table 3.5: Integral division

Operand 1	Operand 2	Quotient	Remainder
+	+	+	+
-	+	-	-
+	-	-	+
-	-	+	-

```
unsigned char a, b, c;
a = b + c;
```

Strictly speaking, this statement requires that the values of `b` and `c` should be promoted to `unsigned int`, the addition performed, the result of the addition cast to the type of `a`, and then the assignment can take place. Even if the result of the `unsigned int` addition of the promoted values of `b` and `c` was different to the result of the `unsigned char` addition of these values without promotion, after the `unsigned int` result was converted back to `unsigned char`, the final result would be the same. An 8-bit addition is more efficient than a 16-bit addition and so the compiler will encode the former.

If, in the above example, the type of a was `unsigned int`, then integral promotion would have to be performed to comply with the ANSI standard.

3.6.2 Shifts applied to integral types

The ANSI standard states that the result of right shifting (>> operator) signed integral types is implementation defined when the operand is negative. Typically, the possible actions that can be taken are that when an object is shifted right by one bit, the bit value shifted into the most significant bit of the result can either be zero, or a copy of the most significant bit before the shift took place. The latter case amounts to a sign extension of the number.

PICC18 performs a sign extension of any `signed` integral type (for example `signed char`, `signed int` or `signed long`). Thus an object with the `signed int` value 0124h shifted right one bit will yield the value 0092h and the value 8024h shifted right one bit will yield the value C012h.

Right shifts of `unsigned` integral values always clear the most significant bit of the result.

Left shifts (<< operator), `signed` or `unsigned`, always clear the least significant bit of the result.

3.6.3 Division and modulus with integral types

The sign of the result of division with integers when either operand is negative is implementation specific. 3.5 shows the expected sign of the result of the division of operand 1 with operand 2 when compiled with PICC18.

In the case where the second operand is zero (division by zero), the result will always be zero.

Table 3.6: Registers Used by the Compiler

Register Name	Description
W	The working register
STATUS	The Status register
PCLAT*x*	Upper holding registers of the program counter*
FSR*x*	Indirect data memory address pointer*
TBLPTR	Indirect program memory address pointer*
PROD	Product result register*
BSR	Bank select register

3.7 Register Usage

The assembly generated from C source code by the compiler will use certain registers that are present on the PIC18 MCU device. Most importantly, the compiler assumes that nothing other than code it generates can alter the contents of these registers. So if the assembly loads a register with a value and no subsequent code generation requires this register, the compiler will assume that the contents of the register are still valid later in the output sequence.

The registers that are special and which are used by the compiler are listed in Table 3.6. Those register starred (*) in the description are multi-byte registers; all components are used by the compiler.

The state of these register must never be changed directly by C code, or by any assembly code in-line with C code. The following example shows a C statement and in-line assembly that violates these rules and changes the ZERO bit in the STATUS register.

```
#include <htc.h>
void getInput(void) {
   ZERO = 0x1; // do not write using C code
   c = read();
#asm
   bcf ZERO_bit ; do not write using in-line assembly code
#endasm
   process(c);
}
```

HI-TECH C is unable to interpret the meaning of in-line assembly code that is encountered in C code. Nor does it associate a variable mapped over an SFR to the actual register itself. Writing to an SFR register using either of these two methods will not flag the register as having changed and may lead to code failure.

3.8 Psects

When the code generator outputs code and data objects, it does so into a number of standard "program sections", referred to as psects. A psect is just a block of something: a block of code; a block of data etc. By having everything inside a psect, all these blocks can be easily recognized and sorted by the linker, even though they have come from different modules.

One of the main jobs of the linker is to group all the psects from the entire project and place these into the available memory for the device.

A psect can be created in assembly code by using the `PSECT` assembler directive (see Section 4.3.10.3). The code generator uses this directive to direct assembly code it produces into the appropriate psect.

3.8.1 Compiler-generated Psects

The code generator places code and data into psects with standard names which are subsequent positioned by the default linker options. The linker does not treat these compiler-generated psects any differently to a psect that has been defined by yourself.

Some psects, in particular the data memory psects, use special naming conventions.

For example, take the `bss` psect. The name `bss` is historical. It holds uninitialized variables. However there may be some uninitialized variables that will need to be located in bank 0 data memory; others may need to be located in bank 1 memory. As these two groups of variables will need to be placed into different memory banks, they will need to be in separate psects so they can be independently controlled by the linker. In addition, the uninitialized variables that are bit variables need to be treated specially so they need their own psect. So there are a number of different psects that all use the same basename, but which have prefixes and suffixes to make them unique.

The general form of these psect names is:

`[bit]`*`psectBaseNameCLASS`*`[`*`div`*`]`

where *`psectBaseName`* is the base name of the psect, such as `bss`.The *`CLASS`* is a name derived from the linker class (see Section 5.7.2) in which the psect will be linked, e.g. `BANK0`. The prefix bit is used if the psect holds `bit` variables. So there may be psects like: `bssBANK0`, `bssBANK1` and `bitbssBANK0` defined by the compiler to hold the uninitialized variables.

If a psect has to be split into two ranges, then the letters `l` (elle) and `h` are used as *`div`* to indicate if it is the lower or higher division. A psect would be split if memory in the middle of a bank has been reserved, or is in some way not available to position objects. If an absolute variable is defined and is located anywhere inside a memory range, that range will need to be split to ensure that anything in the psects located there do not overwrite the absolute object. Thus you might see `bssBANK0l` and `bssBANK0h` psects if a split took place.

The contents of these psects are described below, listed by psect base name.

3.8.1.1 Program Space Psects

checksum This is a psect that is used to mark the position of a checksum that has been requested using the `--CHECKSUM` option, see Section 2.6.20. The checksum value is added after the linker has executed so you will not see the contents of this psect in the assembly list file, nor specific information in the map file.
Linking this psect at a non-default location will have no effect on where the checksum is stored, although the map file will indicate it located at the new address. Do not change the default linker options relating to this psect.

cinit Used by the C initialization runtime startup code. Code in this psect is output by the code generator along with the generated code for the C program (look for it in the project's assembly list file) and does not appear in the runtime startup assembly module.
This psect can be linked anywhere in the program memory, provided they does not interfere with the requirements of other psects. (It should not cross a 4000h address boundary for devices where this is a published errata.)

config Used to store the configuration words.
Do not change the default linker options relating to this psect.

const These psects hold objects that are declared `const` and string literals which are not modifiable. Used when the total amount of const data in a program exceeds 64k.
This psect can be linked anywhere in the program memory provided it does not interfere with the requirements of other psects.

eeprom_data Used to store data to be programmed into the EEPROM data area.
Do not change the default linker options relating to this psect.

end_init Used by initialization code in the `startup.as` module that transfers control to the `main` function. (It should not cross a 4000h address boundary for devices where this is a published errata.)

idata These psects contain the ROM image of any initialised variables. These psects are copied into the `data` psects at startup. In this case, the class name is used to describe the class of the corresponding RAM-based data psect. These psects will be stored in program memory, not the data memory space.
This psect can be linked anywhere in the program memory, provided it does not interfere with the requirements of other psects.

idloc Used to store the ID location words.
Do not change the default linker options relating to this psect.

init Used by initialisation code in the `startup.as` module. This code deals with setting up the target device.
This psect can be linked anywhere in the program memory provided it does not interfere with the requirements of other psects. (It should not cross a 4000h address boundary for devices where this is a published errata.)

intcode Is the psect which contains the executable code for the default or high-priority interrupt service routine. This psect is linked to interrupt vector at address 08H.
Do not change the default linker options relating to this psect. See Section 2.6.24 if you want to move code when using a bootloader.

intcodelo Is the psect which contains the executable code for the low-priority interrupt service routine. This psect is linked to interrupt vector at address 018H.
Do not change the default linker options relating to this psect. See Section 2.6.24 if you want to move code when using a bootloader.

mediumconst These psects hold objects that are declared `const` and string literals which are not modifiable. Used when the total amount of const data in a program exceeds 255 bytes, but does not exceed 64k.
This psect can be linked anywhere in the lower 64k of program memory provided it does not interfere with the requirements of other psects.

powerup This contains executable code for the standard or user-supplied power-up routine.
Do not change the default linker options relating to this psect.

smallconst These psects hold objects that are declared `const` and string literals which are not modifiable. Used when the total amount of const data in a program is less than 255 bytes.
This psect can be linked anywhere in the program memory, provided it does not cross a 100h boundary or interfere with the requirements of other psects.

text*n* Is a `global` psect used for executable code and library functions. *n* is a number. The code associated with each function will be placed a unique text psect.
These psects can be linked anywhere in the program memory, provided they do not interfere with the requirements of other psects. (It should not cross a 4000h address boundary for devices where this is a published errata.)

3.8.1.2 Data Space Psects

bss These psects contain any uninitialized variables. These psects may be linked anywhere in their targeted memory bank and should not overlap access bank memory.

cstack These psects contain the compiled stack. On the stack are `auto`, temporary and parameter variables for the entire program. See Section 3.4.1.1 for information on the compiled stack. These psects may be linked anywhere in their targeted memory bank and should not overlap access bank memory.

data These psects contain the RAM image of any initialized variables. These psects may be linked anywhere in their targeted memory bank and should not overlap access bank memory.

nv These psects are used to store variables qualified `persistent`. They are not cleared or otherwise modified at startup. These psects may be linked anywhere in their targeted memory bank, but should not overlap access bank memory.

3.9 Interrupt Handling in C

The compiler incorporates features allowing interrupts to be handled from C code. Interrupt functions are often called *interrupt service routines* (ISR). Interrupts are also known as *exceptions*. PIC18 devices have two separate interrupt vectors and a priority scheme to dictate how the interrupt code is called.

3.9.1 Interrupt Functions

The function qualifier `interrupt` may be applied to at most two functions to allow them to be called directly from the hardware interrupts. The compiler will process the `interrupt` function differently to any other functions, generating code to save and restore any registers used and exit using the `retfie` instruction instead of a `retlw` or `return` instructions at the end of the function.

(If the `PICC18` option `--STRICT` is used, the interrupt keyword becomes `__interrupt`. Wherever this manual refers to the `interrupt` keyword, assume `__interrupt` if you are using `--STRICT`.)

The PIC18 devices have two interrupts, each with their own vector location. These have different priorities and are known as *low-priority* and *high-priority interrupts*. If the PIC18 is placed in compatibility mode, only one interrupt is available and this defaults to being the high-priority interrupt. An `interrupt` function must be declared as type `interrupt void` and may not have parameters. In addition, the keyword `low_priority` may be used to indicate that the `interrupt` function is to be linked with the low-priority vector when not in compatibility mode. `Interrupt` functions may not be called directly from C code, but they may call other functions itself, subject to certain limitations. Once defined, the corresponding interrupt vector is linked to the `interrupt` function.

An example of a high-priority (default) `interrupt` function is shown here.

```
long tick_count;
void interrupt tc_int(void){
```

```
    ++tick_count;
}
```

A `low-priority interrupt` function may be defined as in the following example.

```
void interrupt low_priority tc_clr(void){
    tick_count = 0;
}
```

It is up to the user to determine and set the priority levels associated with each interrupt source on the PIC18 devices. Defining a low-priority `interrupt` function does *not* put the PIC into interrupt-priority mode.

Low- and high-priority `interrupt` functions have their own separate areas of memory in which to save context, thus a high-priority `interrupt` function may interrupt a low-priority `interrupt` function with no loss of data. The high-priority **interrupt** can also employ the devices' shadow registers to enable rapid context switching during the entry and exit of the service routine.

The `interrupt_level` pragma may be used with either or both `interrupt` functions in the usual way.

3.9.2 Context Switching

The compiler can deal with saving and restoration of the program's state when an interrupt occurs.

3.9.2.1 Context Saving

Some registers are automatically saved by the hardware when an interrupt occurs. Any registers or compiler temporary objects used by the `interrupt` function, other than those saved by the hardware, must be saved in software. This is the context save, or context switch code.

By default, the high-priority `interrupt` function will utilize a fast interrupt switch technique where the W, STATUS and BSR registers are saved and restored via the devices' internal shadow registers. This minimizes code size and reduces the instruction cycles to access the high-priority service routine. Note that for some older devices, the compiler will not use the shadow registers if compiling for the MPLAB ICD debugger, as the debugger itself utilizes these shadow registers.

The compiler fully determines which registers and objects are used by an interrupt function, or any of the functions that it calls (based on the call graph generated by the compiler), and saves these appropriately.

Assembly code placed in-line within the interrupt function is not scanned for register usage. Thus, if you include in-line assembly code into an interrupt function, you may have to add extra assembly code to save and restore any registers or locations used. The same is true for any assembly routines called by the interrupt code.

3.9.2.2 Context Retrieval

Any objects saved by software are automatically restored by software before the interrupt function returns. The order of restoration is the reverse to that used when context is saved.

The `RETFIE` instruction placed at the end of the interrupt code will reload the program counter and the program will return to the location at which it was when the interrupt occurred. If the shadow registers were used to save context, a `RETFIE f` instruction is used to indicate that the contents of the shadow registers should be reloaded to their corresponding register.

3.9.3 Enabling Interrupts

Two macros are available, once you have included `<htc.h>`, which control the masking of all available interrupts. These macros are `ei()`, which enable or unmask all interrupts, and `di()`, which disable or mask all interrupts.

On all PIC18 devices, they affect the GIE bit in the INTCON register. These macros should be used once the appropriate interrupt enable bits for the interrupts that are required in a program have been enabled.

For example:

```
ADIE = 1; // A/D interrupts will be used
PEIE = 1; // all peripheral interrupts are enabled
ei();     // enable all interrupts
// ...
di();     // disable all interrupts
```

Never use this macro to re-enable interrupts inside the interrupt function itself. Interrupts are automatically re-enabled by hardware on execution of the `RETFIE` instruction. Re-enabling interrupts inside an interrupt function may result in code failure.

3.9.4 Function Duplication

It is assumed by the compiler that an interrupt may occur at any time. As all functions are not reentrant (because of the dependence on the compiled stack for local objects, see Section 3.4.1.1), if a function appears to be called by an interrupt function and by main-line code this could normally lead to code failure.

HI-TECH C has a feature which will duplicate the output associated with any function called from more than one call tree in the program's call graph. There will be one call tree associated with main-line code, and one tree for the interrupt function, if defined.

Main-line code will call the original function's output, and the interrupt will call the duplicated function's output. The duplication takes place only in the called function's output; there is no duplication of the C source code itself. The duplicated code and data uses different symbols and are allocated different memory, so are fully independent.

This is similar to the process you would need to undertake if this feature was not implemented in the compiler: the C function could be duplicated by hand, given different names and one called from main-line code; the other from the interrupt function. However, you would have to maintain both functions, and the code would need to be reverted if it was ported to a compiler which did support reentrancy.

The compiler-generated duplicate will have unique identifiers for the assembly symbols used within it. The identifiers consists of the same name used in the original output prefixed with the symbol `i1`.

The output of the function called from main-line code will not use any prefixes and the assembly names will be those normally used.

To illustrate, in a program the function main calls a function called input. This function is also called by an interrupt function.

Examination of the assembly list file will show assembly code for both the original and duplicate function outputs. The output corresponding to the C function `input()` will use the assembly label `_input`. The corresponding label used by the duplicate function will be `i1_input`. If the original function makes reference to a temporary variable, the generated output will use the symbol `??_input`, compared to `??i1_input` for the duplicate output. Even local labels within the function output will be duplicated in the same way. The call graph, in the assembly list file, will show the calls made to both of these functions as if they were independently written. These symbols will also be seen in the map file symbol table.

This feature allows the programmer to use the same source code with compilers that use either reentrant or non-reentrant models. It does not handle cases where functions are called recursively.

Code associated with library functions are duplicated in the same way. This also applies to implicitly called library routines, such as those that perform division or floating-point operations associated with C operators.

3.9.4.1 Disabling Duplication

The automatic duplication of the function may be inhibited by the use of a special pragma.

This should only be done if the source code guarantees that an interrupt cannot occur while the function is being called from any main-line code. Typically this would be achieved by disabling interrupts before calling the function. It is not sufficient to disable the interrupts inside the function after it has been called; if an interrupt occurs when executing the function, the code may fail. See Section 3.9.3 for more information on how interrupts may be enabled and disabled.

The pragma is:

```
#pragma interrupt_level 1
```

The pragma should be placed before the definition of the function that is not to be duplicated. The pragma will only affect the first function whose definition follows.

For example, if the function `read()` is only ever called from main-line code when the interrupts are disabled, then duplication of the function can be prevented if it is also called from an interrupt function as follows.

```
#pragma interrupt_level 1
int read(char device)
{
   // ...
}
```

In main-line code, this function would typically be called as follows:

```
di(); // turn off interrupts
read(IN_CH1);
ei(); // re-enable interrupts
```

3.9.5 Interrupt Registers

It is up to the user how they want the interrupt source configured. All the registers and bits associated with interrupts are defined in the specific header file which can be accessed by including `<htc.h>`. The following is an example of setting up the interrupts associated with the change-on-PORTB source. Interrupt priorities are used and the interrupt source is made a low priority. See your PIC18 datasheet for more information.

```
void main(void) {
   TRISB = 0x80;   // Only RB7 will interrupt on change
   IPEN  = 1;      // Interrupt priorities enabled
   PEIE  = 1;      // enable peripheral interrupts
   RBIP  = 0;      // make this a low priority interrupt
   RBIE  = 1;      // enable PORTB change interrupt
   RBIF  = 0;      // clear any pending events
   GIEL  = 1;      // enable low-priority interrupts
   while(1)continue; // sit here and wait for interrupt
}
void interrupt low_priority b_change(void){
   if(RBIE && RBIF){
```

```
            PORTB;    // Read PORTB to clear any mismatch
            RBIF = 0; // clear event flag
            // process interrupt here
        }
    }
```

3.10 Mixing C and Assembly Code

Assembly code can be mixed with C code using three different techniques. The following section describes writing assembly code in separate assembly modules. The subsequent section looks at two methods of having assembly code being placed in-line with C code.

The following sections describe consideration of mixing Assembly with C code, and some of the special features the compiler uses to allow for assembly-C code interaction.

3.10.1 External Assembly Language Functions

Entire functions may be coded in assembly language as separate `.as` source files, assembled by the assembler, `ASPIC18`, and combined into the binary image using the linker. Assembly source files must not have the same basename os the project name if you are using MPLAB IDE v8.

The following are guidelines that must be adhered to when writing a routine in assembly code that is callable from C code.

- select, or define, a suitable psect for the executable assembly code
- select a name (label) for the routine so that its corresponding C identifier is valid
- ensure that the routine's label is globally accessible, i.e. from other modules
- select an appropriate equivalent C prototype for the routine on which argument passing can be modelled
- optionally, use a signature value to enable type checking of parameters when the function is called
- Limit arguments and return values to single byte-sized objects (Assembly routines may not define variables that reside in the compiled stack. Use global variables for additional arguments.

A mapping is performed on the names of all C functions and non-`static` global variables. See 3.10.3 for a description of mappings between C and assembly identifiers.

TUTORIAL

C-CALLABLE ASSEMBLY ROUTINES The following example goes through the steps of creating an assembly routine. A mapping is performed on the names of all C functions and non-`static` global variables. See Section 3.10.4 for a complete description of mappings between C and assembly identifiers.

An assembly routine is required which can add an 8-bit quantity passed to the routine with the contents of PORTB and return this as an 8-bit quantity.

Most compiler-generated executable code is placed in psects called `text`*n*, where *n* is a number. (see Section 3.8.1). We will create our own text psect based on the psect the compiler uses. Check the assembly list file to see how the text psects normally appear. You may see a psect such as the following generated by the code generator.

```
PSECT text0,local,class=CODE,reloc=2
```

TUTORIAL

See Section 4.3.10.3 for detailed information on the flags used with the `PSECT` assembler directive. This psect is called `text0`. It is flagged `local`, which means that it is distinct from other psects with the same name. It lives in the `CODE` class. This flag is important as it means it will be automatically placed in the area of memory set aside for code. With this flag in place, you do not need to adjust the default linker options to have the psect correctly placed in memory. The last option, the `reloc` value, is also very important. This indicates that the psect must start on an even address boundary. The PIC18 program memory space is byte addressable and instructions must be aligned on an even address.

We simply need to choose a different name, so we might choose the name `mytext`, as the psect name in which we will place out routine, so we have:

```
PSECT mytext,local,class=CODE,reloc=2
```

TUTORIAL

Let's assume we would like to call this routine `add` in the C domain. In assembly domain we must choose the name `_add` as this then maps to the C identifier `add`. If we had chosen `add` as the assembly routine, then it could never be called from C code. The name of the assembly routine is the label that we will place at the beginning of the assembly code. The label we would use would look like this.

```
_add:
```

TUTORIAL

We need to be able to call this from other modules, some make this label globally accessible, by using the GLOBAL assembler directive (Section 4.3.10.1).

```
GLOBAL _add
```

TUTORIAL

By compiling a dummy C function with a similar prototype to this assembly routine, we can determine the signature value. The C-equivalent prototype to this routine would look like:

```
char add(char);
```

TUTORIAL

Check the assembly list file for the signature value of such a function. Signature values are not mandatory, but allow for additional type checking to be made by the linker. We determine that the following SIGNAT directive (Section 4.3.10.21) can be used.

```
SIGNAT _add,4217
```

TUTORIAL

The W register will be used for passing in the argument.

Here is an example of the complete routine which could be placed into an assembly file and added to your project. The GLOBAL and SIGNAT directives do not generator code, and hence do not need to be inside the mytext psect, although you can place them there if you prefer. The BANKSEL directive and BANKMASK macro have been used to ensure that the correct bank was selected and that all addresses are masked to the appropriate size.

```
#include <pic18.inc>
GLOBAL _add      ; make _add globally accessible
SIGNAT _add,4217 ; tell the linker how it should be called
; everything following will be placed into the mytext psect
PSECT mytext,local,class=CODE,delta=2
; our routine to add WREG to PORTB and return the result
; W is loaded by the calling function
_add:
   BANKSEL (PORTB)          ; select the bank of this object
   ADDWF BANKMASK(PORTB),w  ; add parameter to port
   ; the result is already in the required location (W)
   ; so we can just return immediately
   RETURN
```

TUTORIAL

To compile this, the assembly file must be preprocessed as we have used the C preprocessor `#include` directive. See Section 2.6.12.

To call an assembly routine from C code, a declaration for the routine must be provided. This ensures that the compiler knows how to encode the function call in terms of parameters and return values.

Here is a C code snippet that declares the operation of the assembler routine, then calls the routine.

```
// declare the assembly routine so it can be correctly called
extern unsigned char add(unsigned char a);
void main(void) {
   volatile unsigned char result;
   a = read_port();
   result = add(5);   // call the assembly routine
}
```

3.10.2 #asm, #endasm and asm()

PIC18 instructions may also be directly embedded "in-line" into C code using the directives `#asm`, `#endasm` or the statement `asm()`.

The `#asm` and `#endasm` directives are used to start and end a block of assembly instructions which are to be embedded into the assembly output of the code generator. The `#asm` and `#endasm` construct

is not syntactically part of the C program, and thus it does not obey normal C flow-of-control rules, however you can easily include multiple instructions with this form of in-line assembly.

The `asm()` statement is used to embed a single assembler instruction. This form looks and behaves like a C statement, however each instruction must be encapsulated within an `asm()` statement.

You should not use a `#asm` block within any C constructs such as `if`, `while`, `do` etc. In these cases, use only the `asm("")` form, which is a C statement and will correctly interact with all C flow-of-control structures.

The following example shows both methods used to rotate a byte left through carry:

```
unsigned char var;
void main(void){
   var = 1;
#asm   // like this...
   movlb (_var) >> 8
   rlcf  (_var)&0ffh,f
#endasm
   asm("movlb (_var)>>8");
   asm("rlcf (_var)&0ffh,f");
}
```

When using in-line assembly code, great care must be taken to avoid interacting with compiler-generated code. If in doubt, compile your program with the `PICC18 -S` option and examine the assembly code generated by the compiler.

IMPORTANT NOTE: the `#asm` and `#endasm` construct is not syntactically part of the C program, and thus it does *not* obey normal C flow-of-control rules. For example, you cannot use a `#asm` block with an if statement and expect it to work correctly. If you use in-line assembler around any C constructs such as `if`, `while`, `do` etc. then you should use only the `asm("")` form, which is a C statement and will correctly interact with all C flow-of-control structures.

3.10.3 Accessing C objects from within Assembly Code

Global C objects may be directly accessed from within assembly code using their name prepended with an *underscore* character. For example, the object `foo` defined globally in a C module:

```
int foo;
```

may be access from assembler as follows.

```
GLOBAL  _foo
movwf   _foo
```

If the assembler is contained in a different module, then the `GLOBAL` assembler directive should be used in the assembly code to make the symbol name available, as above. If the object is being accessed from in-line assembly in another module, then an `extern` declaration for the object can be made in the C code, for example:

```
extern int foo;
```

This declaration will only take effect in the module if the object is also accessed from within C code. If this is not the case then, an in-line `GLOBAL` assembler directive should be used. Care should be taken if the object is defined in a bank other than 0. The address of a C object includes the bank information which must be stripped before the address can be used in most PIC18 instructions. The exceptions are the `movff` and `lsfr` instructions. Failure to do this may result in fixup errors issued by the linker. If in doubt as to writing assembler which access C objects, write code in C which performs a similar task to what you intend to do and study the assembler listing file produced by the compiler.

C identifiers are assigned different symbols in the output assembly code so that an assembly identifier cannot conflict with an identifier defined in C code. If assembly programmers choose identifier names that do not begin with an *underscore*, these identifiers will never conflict with C identifiers. Importantly, this implies that the assembly identifier, `i`, and the C identifier `i` relate to different objects at different memory locations.

3.10.3.1 Accessing special function register names from assembler

If writing separate assembly modules, SFR definitions will not automatically be accessible.

The assembly header file `<pic18.inc>` can be used to gain access to these register definitions from separate assembly modules. Do not use this header for assembly in-line with C code as it will clash with definitions in `<htc.h>`.

Include the file using the assembler's `INCLUDE` directive, (see Section 4.3.11.4) or use the C preprocessor's `#include` directive (see Section 3.11.2). If you are using the latter method, make sure you compile with the `-P` driver option to preprocess assembly files, see Section 2.6.12.

The symbols in this header file look similar to the identifiers used in the C domain when including <htc.h>, e.g. PORTA, EECON1 etc. They are different symbols in different domains, but will map to the same memory location.

Bits within registers are defined as the *registerName*,*bitNumber*. So for example, RA0 is defined as PORTA,0.

Here is an example of an assembly module that uses SFRs.

```
#include <pic18.inc>
GLOBAL _setports
PSECT text,class=CODE,local,reloc=2
_setports:
    MOVLW 0xAA
    BANKSEL (PORTA)
    MOVWF BANKMASK(PORTA)
    BANKSEL (PORTB)
    BSF RB1
```

If you wish to access register definitions from assembly that is in-line with C code, different definitions need to be used, but these are already available once you include the <htc.h> header file for the C part of the module.

The symbols used for register names will be the same as those defined by <pic18.inc>; however, the names assigned to bit variables within the registers will include the suffix _bit. So for example, the example given previously could be rewritten as in-line assembly as follows.

```
#include <htc.h>
#asm MOVLW 0xAA
    BANKSEL (PORTA)
    MOVWF BANKMASK(PORTA)
    BANKSEL (PORTB)
    BSF RB1_bits
#endasm
```

Care must be taken to ensure that you do not destroy the contents of registers that are holding intermediate values of calculations. Some registers are used by the compiler and writing to these registers directly can result in code failure. The compiler does not detect when SFRs have changed as a result of assembly code that writes to them directly. The list of registers used by the compiler and further information can be found in Section 3.7.

3.10.4 Interaction between Assembly and C Code

HI-TECH C Compiler for PIC18 MCUs incorporates several features designed to allow C code to obey requirements of user-defined assembly code.

The command-line driver ensures that all user-defined assembly files have been processed first, before compilation of C source files begin. The driver is able to read and analyse certain information in the relocatable object files and pass this information to the code generator. This information is used to ensure the code generator takes into account requirement of the assembly code.

3.10.4.1 Absolute Psects

Some of the information that is extracted from the relocatable objects by the driver relates to absolute psects, specifically psects defined using the `abs` and `ovrld`, `PSECT` flags, see Section 4.3.10.3 for more information. These are psects have been rarely required in general coding, but do allow for data to be collated over multiple modules in a specific order.

HI-TECH C Compiler for PIC18 MCUs is able to determine the address bounds of absolute psects to enure that the output of C code does not consume specific resources required by the assembly code. The code generator will ensure that any memory used by these psects are reserved and not used by C code. The linker options are also adjusted by the driver to enure that this memory is not allocated.

TUTORIAL

PROCESSING OF ABSOLUTE PSECTS An assembly code files defines a table that must be located at address 210h in the data space. The assembly file contains:

```
PSECT lkuptbl,class=RAM,space=1,abs,ovlrd
ORG 210h
lookup:
ds 20h
```

When the project is compiled, this file is assembled and the resulting relocatable object file scanned for absolute psects. As this psect is flagged as being `abs` and `ovlrd`, the bounds and space of the psect will be noted — in this case a memory range from address 210h to 22fh in memory space 1 is being used. This information is passed to the code generator to ensure that these address spaces are not used by C code. The linker will also be told to remove these ranges from those available, and this reservation will be observable in the map file. The RAM class definition, for example, may look like:

```
-ARAM=00h-0FFhx2,0200h-020Fh,0230h-02FFh,0300h-03FFhx3
```

for an 18F452 device, showing that addresses 210h through 22F were reserved from this class range.

3.10.4.2 Undefined Symbols

Variables can be defined in assembly code if required, but in some instances it is easier to do so in C source code, in other cases, the symbols may need to be accessable from both assembly and C source code.

A problem can occur if there is a variable defined in C code, but is never referenced throughout the entire the C program. In this case, the code generator may remove the variable believing it is unused. If assembly code is relying on this definition an error will result.

To work around this issue, HI-TECH C Compiler for PIC18 MCUs also searches assembly-derived object files for symbols which are undefined. These will typically be symbols that are used, but not defined, in assembly code. The code generator is informed of these symbols, and if they are encountered in the C code the variable is automatically marked as being volatile. This is the equivalent of the programmer having qualified the variable as being `volatile` in the source code, see Section 3.3.10. Variables qualified as `volatile` will never be removed by the code generator, even if they appear to be unused throughout the program.

TUTORIAL

PROCESSING OF UNDEFINED SYMBOLS A C source module defines a global variable as follows:

```
int input;
```

but this variable is only ever used in assembly code. The assembly module(s) can simply declare and link in to this symbol using the `GLOBAL` assembler directive, and them make use of the symbol.

```
GLOBAL _input
PSECT text,class=CODE,reloc=2
movff PORTA,_input
```

In this instance the C variable `input` will not be removed and be treated as if it was qualified `volatile`.

3.11 Preprocessing

All C source files are preprocessed before compilation. Assembler files can also be preprocessed if the `-P` command-line option is issued, see Section 2.6.12.

3.11.1 C Language Comments

HI-TECH C Compiler for PIC18 MCUs accepts both block and in-line (C99 standard) C source comments, as shown in the following examples. In-line comments are normally termined by the *newline* character, however they can span multiple lines when the line is terminated with a *backslash* character.

```
/* I am a block comment
that can run over more
than one line of source */
// I am an in-line comment
// I am an in-line comment \
  that spans two lines
```

Both these comment styles can be used, in addition to the standard assembly comment (see Section 4.3.5), in assembly source code if the `-P` command-line option is issued, see Section 2.6.12.

All comments are removed by the C preprocessor before being passed to the parser application.

3.11.2 Preprocessor Directives

HI-TECH C Compiler for PIC18 MCUs accepts several specialised preprocessor directives in addition to the standard directives. All of these are listed in Table 3.7.

Macro expansion using arguments can use the `#` character to convert an argument to a string, and the `##` sequence to concatenate tokens.

3.11.3 Predefined Macros

The compiler drivers define certain symbols to the preprocessor (`CPP`), allowing conditional compilation based on chip type etc. The symbols listed in Table 3.8 show the more common symbols defined by the drivers. Each symbol, if defined, is equated to 1 unless otherwise stated.

Symbol	When set	Usage
`HI_TECH_C`	When not in C18 compatibility mode	To indicate that the compiler in use is HI-TECH C.
`_HTC_EDITION_`	Always	To indicate which of PRO, STD or Lite _HTC_EDITION_ compiler is in use. Values of 2, 1 or 0 are assigned respectively.
continued...		

_HTC_VER_MAJOR_	Always	To indicate the integer component of the compiler's version number.
_HTC_VER_MINOR_	Always	To indicate the decimal component of the compiler's version number.
_HTC_VER_PATCH_	Always	To indicate the patch level of the compiler's version number.
__PICC18__	When not in C18 compatibility mode	To indicate the use of a HI-TECH PICC-18 compiler.
MPC	When not in C18 compatibility mode	To indicate the code is compiled for the Microchip PIC family.
_PIC18	When not in C18 compatibility mode	To indicate that this is a PIC18 device.
_ROMSIZE	Always	To indicate the number of bytes of program space this device has.
_RAMSIZE	Always	To indicate the number of bytes of data space this device has.
_EEPROMSIZE	If EEPROM is present	To indicate if EEPROM memory is present and how many bytes are available.
_FLASH_ERASE_SIZE	Always	To indicate the number of bytes erased in a single flash-erase operation at runtime.
_FLASH_WRITE_SIZE	Always	To indicate the number of bytes erased in a single flash-write operation at runtime.
__MPLAB_REALICE__	--DEBUGGER=REALICE	To indicate that the code is being generated for the Microchip Realice debugger.
__MPLAB_PICKIT2__	--DEBUGGER=PICKIT2	To indicate that the code is being generated for the Microchip PICKIT2 debugger.
__MPLAB_PICKIT3__	--DEBUGGER=PICKIT3	To indicate that the code is being generated for the Microchip PICKIT3 debugger.
__MPLAB_ICD__	--DEBUGGER=ICD2/ICD3	To indicate that the code is being generated for the Microchip ICD In-Circuit debugger. Value is 2 or 3.
_ICDROM_START	--DEBUGGER=ICD2/ICD3	Defined the start address of the ICD's reserved program space
_ICDROM_END	--DEBUGGER=ICD2/ICD3	Defined the end address of the ICD's reserved program space

continued...

_ERRATA_TYPES	Always	Defines a bitmask to show which types of silicon errata may be applicable to this build.
errata_type	When errata workaround is employed	Defined when the errata workaround is applied, e.g. ERRATA_4000_BOUNDARY
__OPTIMIZE_*type*__	When optimization is enabled	To indicate if speed or space optimizations are enabled with __OPTIMIZE_SPEED__ or __OPTIMIZE_SPACE__
_chipname	When chip selected	To indicate the specific chip type selected, e.g. _18F452
__chipname	When chip selected	To indicate the specific chip type selected, e.g. __18F452
*_device*_FAMILY_	Always	To indicate the device family grouping, as determined by the device INI file, e.g. _18FXX2_FAMILY
__J_PART	When compiling for a 'J' device	To indicate target device has a 'J' in its name.
__18CXX	When in C18 compatibility mode	To indicate the compiler is operating in C18 compatibility mode
__TRADITIONAL18__	When in C18 compatibility mode and using the tradition instruction set	To indicate the compiler is operating in C18 compatibility mode and the extended instruction set is not enabled.
__EXTENDED18__	When in C18 compatibility mode and using the extended instruction set	To indicate the compiler is operating in C18 compatibility mode and the extended instruction set is enabled.
__SMALL__	When in C18 compatibility mode and the memory model is small	To indicate the compiler is operating in C18 compatibility mode and the small memory model is being used.
__LARGE__	When in C18 compatibility mode and the memory model is large	To indicate the compiler is operating in C18 compatibility mode and the large memory model is being used.
__FILE__	Always	To indicate this source file being preprocessed.
__LINE__	Always	To indicate this source line number.
continued...		

__DATE__	Always	To indicate the current date, e.g. May 21 2004
__TIME__	Always	To indicate the current time, e.g. 08:06:31.
_PLIB	When the peripheral libraries are linked in	To indicate that the microchip compatible peripheral libraries have been linked in.

3.11.4 Pragma Directives

There are certain compile-time directives that can be used to modify the behaviour of the compiler. These are implemented through the use of the ANSI standard #pragma facility. The format of a pragma is:

```
#pragma keyword options
```

where *keyword* is one of a set of keywords, some of which are followed by certain *options*. A list of the keywords is given in Table 3.9. Those keywords not discussed elsewhere are detailed below.

3.11.4.1 The #pragma printf_check Directive

Certain library functions accept a format string followed by a variable number of arguments in the manner of printf(). Although the format string is interpreted at runtime, it can be compile-time checked for consistency with the remaining arguments.

This directive enables this checking for the named function, e.g. the system header file <stdio.h> includes the directive #pragma printf_check(printf) const to enable this checking for printf(). You may also use this for any user-defined function that accepts printf-style format strings. The qualifier following the function name is to allow automatic conversion of pointers in variable argument lists. The above example would cast any pointers to strings in RAM to be pointers of the type (const char *)

Note that the warning level must be set to -1 or below for this option to have any visible effect. See Section 2.6.64.

Table 3.7: Preprocessor directives

Directive	Meaning	Example
`#`	preprocessor null directive, do nothing	`#`
`#assert`	generate error if condition false	`#assert SIZE > 10`

`#asm`	signifies the beginning of in-line assembly	`#asm`
		`mov r0, r1h`
		`#endasm`

`#define`	define preprocessor macro	`#define SIZE 5`
		`#define FLAG`
		`#define add(a,b) ((a)+(b))`

`#elif`	short for `#else #if`	`see #ifdef`
`#else`	conditionally include source lines	`see #if`
`#endasm`	terminate in-line assembly	`see #asm`
`#endif`	terminate conditional source inclusion	`see #if`
`#error`	generate an error message	`#error Size too big`

`#if`	include source lines if constant expression true	`#if SIZE < 10`
		`c = process(10)`
		`#else`
		`skip();`
		`#endif`

`#ifdef`	include source lines if preprocessor symbol defined	`#ifdef FLAG`
		`do_loop();`
		`#elif SIZE == 5`
		`skip_loop();`
		`#endif`

`#ifndef`	include source lines if preprocessor symbol not defined	`#ifndef FLAG`
		`jump();`
		`#endif`

`#include`	include text file into source	`#include <stdio.h>` `#include "project.h"`
`#line`	specify line number and filename for listing	`#line 3 final`
`#`*nn*	(where *nn* is a number) short for `#line` *nn*	`#20`
`#pragma`	compiler specific options	`3.11.4`
`#undef`	undefines preprocessor symbol	`#undef FLAG`
`#warning`	generate a warning message	`#warning Length not set`

Table 3.9: Pragma directives

Directive	Meaning	Example
`printf_check`	Enable printf-style format string checking	`#pragma printf_check(printf) const`
`regsused`	Specify registers which are used by a function	`#pragma regsused _func r4`
`switch`	Specify code generation for switch statements	`#pragma switch direct`
`inline`	Specify function is inline	`#pragma inline (_delay)`
`interrupt_level`	disable function at the specified level	`#pragma interrupt_level 1`
`warning`	Control messaging parameters	`#pragma warning disable 299,407`

3.11.4.2 The #pragma regsused Directive

The `#pragma regsused` directive allows the programmer to indicate register usage for functions that will not be "seen" by the code generator, for example if they were written in assembly code. It has no effect when used with functions defined in C code, but in these cases the register usage of these functions can be accurately determined by the compiler and the pragma is not required.

The compiler will determine only those registers and objects which need to be saved for an interrupt function defined and use of this pragma allows the code generator to also determine register usage for routines written in assembly code.

The general form of the pragma is:

```
#pragma regsused routineName registerList
```

where *routineName* is the C equivalent name of the function or routine whose register usage is being defined, and *registerList* is a space-separated list of registers names, as shown in Table 3.10. The pragma must be placed after the declaration for the assembly routine whose register usage is being specified.

Those registers not listed are assumed to be unused by the function or routine. The code generator may use any of these registers to hold values across a function call. Hence, if the routine does in fact use these registers, unreliable program execution may eventuate.

The register names are not case sensitive and a warning will be produced if the register name is not recognized. A blank list indicates that the specified function or routine uses no registers.

Table 3.10: Valid register names

Register Name	Description
wreg	W register
status	STATUS register
pclat	PCLATH register
prodl, prodh	product result registers
fsr0, fsr1, fsr2	indirect data pointers 0, 1 and 2
tblptrl, tblptrh, tblptru	table pointer registers

Table 3.11: Switch types

switch type	description
speed	Use the faster switch method
space	Use the smallest code size method
time	Use a fixed delay switch method
auto	use smallest code size method (default)

For example, a routine called _search is written in assembly code. In the C source, we may write:

```
extern void search(void);
#pragma regsused search wreg status fsr0
```

to indicate that this routine used the W register, STATUS and FSR0.

3.11.4.3 The #pragma switch Directive

Normally, the compiler chooses how switch statements will be encoded to produce the smallest possible code size. The #pragma switch directive can be used to force the compiler to use a different coding strategy.

The general form of the switch pragma is:

```
#pragma switch switch_type
```

where *switch_type* is one of the available switch methods listed in Table 3.11.

Specifying the time option to the #pragma switch directive forces the compiler to generate the table look-up style switch method. This is mostly useful where timing is an issue for switch statements (i.e.: state machines).

This pragma affects all subsequent code. The auto option may be used to revert to the default behaviour.

3.11.4.4 The #pragma inline Directive

The `#pragma inline` directive is used to indicate to the compiler that a function will be inlined. The directive is only usable with special functions that the code generator will handle specially, e.g the `_delay` function.

This pragma should not be used with user-defined functions; the code generator must be aware of how to generator code for those functions specified as inline.

3.11.4.5 The #pragma interrupt_level Directive

The `#pragma interrupt_level` directive can be used to disable function duplication if it is called from both interrupt and main-line code. See 3.9.4.1 for ore information and examples of its operation.

3.11.4.6 The #pragma warning Directive

The warning disable pragma Some warning messages can be disabled by using the `warning disable` pragma. This pragma will only affect warnings that are produced by either parser or the code generator, i.e. errors directly associated with C code. The position of the pragma is only significant for the parser, i.e. a parser warning number may be disabled, then re-enabled around a section of the code to target specific instances of the warning. Specific instances of a warning produced by the code generator cannot be individually controlled. The pragma will remain in force during compilation of the entire module.

The state of those warnings which have been disabled can preserved and recalled using the `warning push` and `warning pop` pragmas. Pushes and pops can be nested to allow a large degree of control over the message behaviour.

TUTORIAL

DISABLING A WARNING The following example shows the warning associated with qualifying an `auto` object being disabled, number 348.

```
void main(void)
{
#pragma warning disable 348
near int c;
#pragma warning enable 348
```

```
/* etc */
}
int rv(int a)
{
near int c;
/* etc */
}
```

which will issue only one warning associated with the second definition of the auto variable `c`. Warning number 348 is disabled during parsing of the definition of the auto variable, `c`, inside the function `main`.

```
altst.c:  35:  (348) auto variable "c" should not be qualified (warning)
```

This same affect would be observed using the following code.

```
void main(void)
{
#pragma warning push
#pragma warning disable 348
near int c;
#pragma warning pop
/* etc */
}
int rv(int a)
{
near int c;
/* etc */
}
```

Here the state of the messaging system is saved by the `warning push` pragma. Warning 348 is disabled, then after the source code which triggers the warning, the state of the messaging system is retrieved by the use of the `warning pop` pragma.

The warning error/warning pragma It is also possible to change the type of some messages. This is only possible by the use of the `warning pragma` and only affects messages generated by the parser or code generator. The position of the pragma is only significant for the parser, i.e. a parser message number may have its type changed, then reverted back around a section of the code to target specific instances of the message. Specific instances of a message produced by the code generator cannot be individually controlled. The pragma will remain in force during compilation of the entire module.

TUTORIAL

The following shows the warning produced in the previous example being converted to an error for the instance in the function `main()`.

```
void main(void)
{
#pragma warning error 348
near int c;
#pragma warning warning 348
/* etc */
}
int rv(int a)
{
near int c;
/* etc */
}
```

Compilation of this code would result in an error, and as with any error, this will force compilation to cease after the current module has concluded, or the maximum error count has been reached.

3.12 Linking Programs

The compiler will automatically invoke the linker unless requested to stop after producing assembly code (`PICC18 -S` option) or object code (`PICC18 -C` option).

HI-TECH C, by default, generates Intel HEX. Use the `--OUTPUT=` option to specify a different output format.

After linking, the compiler will automatically generate a memory usage map which shows the address used by, and the total sizes of, all the psects which are used by the compiled code.

The program statistics shown after the summary provides more concise information based on each memory area of the device. This can be used as a guide to the available space left in the device.

More detailed memory usage information, listed in ascending order of individual psects, may be obtained by using the `PICC18 --SUMMARY=psect` option. Generate a map file for the complete memory specification of the program.

3.12.1 Replacing Library Modules

Although HI-TECH C comes with a librarian (LIBR) which allows you to unpack a library files and replace modules with your own modified versions, you can easily replace a library module that is linked into your program without having to do this. If you add the source file which contains the library routine you wish to replace on the command-line list of source files then the routine will replace the routine in the library file with the same name.

This method works due to the way the linker scans source and library file. When trying to resolve a symbol (in this instance a function name) the linker first scans all source modules for the definition. Only if it cannot resolve the symbol in these files does it then search the library files. Even though the symbol may be defined in a source file and a library file, the linker will not search the libraries and no multiply defined symbol error will result. This is not true if a symbol is defined twice in source files.

For example, if you wished to make changes to the library function `max()` which resides in the file `max.c` in the SOURCES directory, you could make a copy of this source file, make the appropriate changes and then compile and use it as follows.

```
PICC18 --chip=18F242 main.c init.c max.c
```

The code for `max()` in `max.c` will be linked into the program rather than the `max()` function contained in the standard libraries. Note, that if you replace an assembler module, you may need the `-P` option to preprocess assembler files as the library assembler files often contain C preprocessor directives.

3.12.2 Signature Checking

The compiler automatically produces signatures for all functions. A signature is a 16-bit value computed from a combination of the function's return data type, the number of its parameters and other information affecting the calling sequence for the function. This signature is output in the object code of any function referencing or defining the function.

At link time the linker will report any mismatch of signatures. HI-TECH C Compiler for PIC18 MCUs is only likely to issue a mismatch error from the linker when the routine is either a precompiled object file or an assembly routine. Other function mismatches are reported by the code generator.

TUTORIAL

It is sometimes necessary to write assembly language routines which are called from C using an `extern` declaration. Such assembly language functions should include a signature which is compatible with the C prototype used to call them. The simplest method of determining the correct signature for a function is to write a dummy C function with the same prototype and compile it to assembly language using the PICC18 `-S` option. For example, suppose you have an assembly language routine called `_widget` which takes two `int` arguments and returns a `char` value. The prototype used to call this function from C would be:

```
extern char widget(int, int);
```

Where a call to `_widget` is made in the C code, the signature for a function with two `int` arguments and a `char` return value would be generated. In order to match the correct signature the source code for widget needs to contain an assembler SIGNAT pseudo-op which defines the same signature value. To determine the correct value, you would write the following code:

```
char widget(int arg1, int arg2)
{
}
```

and compile it to assembly code using

```
PICC18 -S x.c
```

The resultant assembly code includes the following line:

```
SIGNAT _widget,8249
```

The SIGNAT pseudo-op tells the assembler to include a record in the `.obj` file which associates the value 8249 with symbol `_widget`. The value 8249 is the correct signature for a function with two `int` arguments and a `char` return value. If this line is copied into the `.as` file where `_widget` is defined, it will associate the correct signature with the function and the linker will be able to check for correct argument passing. For example, if another `.c` file contains the declaration:

```
extern char widget(long);
```

then a different signature will be generated and the linker will report a signature mismatch which will alert you to the possible existence of incompatible calling conventions.

Table 3.12: Supported standard I/O functions

Function name	Purpose
`printf(const char * s, ...)`	Formatted printing to `stdout`
`sprintf(char * buf, const char * s, ...)`	Writes formatted text to `buf`

3.12.3 Linker-Defined Symbols

The link address of a psect can be obtained from the value of a global symbol with name __L*name* where *name* is the name of the psect. For example, __Lbss is the low bound of the `bss` psect. The highest address of a psect (i.e. the link address plus the size) is symbol __H*name*.

If the psect has different load and link addresses the load start address is specified as __B*name*.

3.13 Standard I/O Functions and Serial I/O

A number of the standard I/O functions are provided, specifically those functions intended to read and write formatted text on standard output and input. A list of the available functions is in Table 3.12. More details of these functions can be found in Appendix A.

Before any characters can be written or read using these functions, the `putch()` (for `printf()`) and `getch()` (for `scanf()`) functions must be written to define `stdin` and `stdout`, respectively. Other routines which may be required include `getche()` and `kbhit()`.

You will find samples of serial code which implements the `putch()` and `getch()` functions in the file serial.c in the `SAMPLES` directory.

Chapter 4

Macro Assembler

The Macro Assembler included with HI-TECH C Compiler for PIC18 MCUs assembles source files for PIC18 MCUs. This chapter describes the usage of the assembler and the directives (assembler pseudo-ops and controls) accepted by the assembler in the source files.

The HI-TECH C Macro Assembler package includes a linker, librarian, cross reference generator and an object code converter.

Athough the term "assembler" is almost universally used to decribe the tool which converts human-readable mnemonics into machine code, both "assembler" and "assembly" are used to describe the source code which such a tool reads. The latter is more common and is used in this manual to describe the language. Thus you will see the terms *assembly language* (or just *assembly*), *assembly listing* and etc, but *assembler options*, *assembler directive* and *assembler optimizer*.

4.1 Assembler Usage

The assembler is called `ASPIC18` and is available to run on *Windows*, *Linux* and *Mac OS* systems. Note that the assembler will not produce any messages unless there are errors or warnings — there are no "assembly completed" messages.

Typically the command-line driver, PICC18, is used to envoke the assembler as it can be passed assembler source files as input, however the options for the assembler are supplied here for instances

where the assembler is being called directly, or when they are specified using the command-line driver option `--SETOPTION`, see Section 2.6.58.

The usage of the assembler is similar under all of available operating systems. All command-line options are recognised in either upper or lower case. The basic command format is shown:

```
ASPIC18 [ options ] files
```

files is a space-separated list of one or more assembler source files. Where more than one source file is specified the assembler treats them as a single module, i.e. a single assembly will be performed on the concatenation of all the source files specified. The files must be specified in full, no default extensions or suffixes are assumed.

options is an optional space-separated list of assembler options, each with a *minus sign* - as the first character. A full list of possible options is given in Table 4.1, and a full description of each option follows.

Table 4.1: ASPIC18 command-line options

Option	Meaning	Default
-A	Produce assembler output	Produce object code
-C	Produce cross-reference file	No cross reference
-C*chipinfo*	Define the chipinfo file	dat\picc-18.ini
-E[*file*|*digit*]	Set error destination/format	
-F*length*	Specify listing form length	66
-H	Output hex values for constants	Decimal values
-I	List macro expansions	Don't list macros
-L[*listfile*]	Produce listing	No listing
-O	Perform optimization	No optimization
-O*outfile*	Specify object name	srcfile.obj
-P*processor*	Define the processor	
-R	Specify non-standard ROM	
-T*width*	Specify listing page width	80
-V	Produce line number info	No line numbers
-W*level*	Set warning level threshold	0
-X	No local symbols in OBJ file	

4.2 Assembler Options

The command-line options recognised by ASPIC18 are as follows.

-A An assembler file with an extension `.opt` will be produced if this option is used. This is useful when checking the optimized assembly produced using the `-O` assembler option. Thus if both `-A` and `-O` are used with an assembly source file, the file will be optimized and rewritten, without the usual conversion to an object file.
The output file, when this option is used, is a valid assembly file that can be passed to the assembler. This differs to the assembly list file produced by the assembler when the -L assembler option is used.

-C A cross reference file will be produced when this option is used. This file, called *`srcfile`*`.crf`, where *`srcfile`* is the base portion of the first source file name, will contain raw cross reference information. The cross reference utility `CREF` must then be run to produce the formatted cross reference listing. See Section 4.7 for more information.

-C*chipinfo* Specify the chipinfo file to use. The chipinfo file is called `picc-18.ini` and can be found in the `DAT` directory of the compiler distribution.

-E[*file*|*digit*] The default format for an error message is in the form:

```
filename: line: message
```

where the error of type *`message`* occurred on line *`line`* of the file *`filename`*.
The `-E` option with no argument will make the assembler use an alternate format for error and warning messages. Use of the option in this form has a similar effect as the same option used with command-line driver. See Section 2.5 for more information.
Specifying a digit as argument has a similar effect, only it allows selection of pre-set message formats.
Specifying a filename as argument will force the assembler to direct error and warning messages to a file with the name specified.

-F*length* By default when an assembly list file is requested (see assembler option `-L`), the listing format is pageless, i.e. the assembly listing output is continuous. The output may be formatted into pages of varying lengths. Each page will begin with a header and title, if specified. The `-F` option allows a page length to be specified. A zero value of *`length`* implies pageless output. The length is specified in a number of lines.

-H Particularly useful in conjunction with the `-A` or `-L ASPIC18` options, this option specifies that output constants should be shown as hexadecimal values rather than decimal values.

-I This option forces listing of macro expansions and unassembled conditionals which would otherwise be suppressed by a `NOLIST` assembler control. The `-L` option is still necessary to produce a listing.

-L*listfile* This option requests the generation of an assembly listing file. If *listfile* is specified then the listing will be written to that file, otherwise it will be written to the standard output. An assembly listing file contains additional fields, such as the address and opcode fields, which are not part of the assembly source syntax, hence these files cannot be passed to the assembler for compilation. See the assembler -A option for generating processed assembly source files that can be used as source files in subsequent compilation.

-O This requests the assembler to perform optimization on the assembly code. Note that the use of this option slows the assembly process down, as the assembler must make an additional pass over the input code. Debug information for assembler code generated from C source code may become unreliable.

-O*outfile* By default the assembler determines the name of the object file to be created by stripping any suffix or extension (i.e. the portion after the last dot) from the first source filename and appending .obj. The -O option allows the user to override the default filename and specify a new name for the object file.

-P*processor* This option defines the processor which is being used. The processor type can also be indicated by use of the PROCESSOR directive in the assembler source file, see Section 4.3.10.20. You can also add your own processors to the compiler via the compiler's chipinfo file.

-T*width* This option allows specification of the assembly list file width, in characters. *width* should be a decimal number greater than 41. The default width is 80 characters.

-V This option will include line number and filename information in the object file produced by the assembler. Such information may be used by debuggers. Note that the line numbers will correspond with assembler code lines in the assembler file. This option should not be used when assembling an assembler file produced by the code generator from a C source file, i.e. it should only be used with hand-written assembler source files.

-W[!]*warnlevel* This option allow the warning threshold level to be set. This will limit the number of warning messages produce when the assembler is executing. The effect of this option is similar to the command-line driver's --WARN option, see Section 2.6.64. See Section 2.5 for more information.

-X The object file created by the assembler contains symbol information, including local symbols, i.e. symbols that are neither public or external. The -X assembler option will prevent the local symbols from being included in the object file, thereby reducing the file size.

4.3 HI-TECH C Assembly Language

The source language accepted by the macro assembler, `ASPIC18`, is described below. All opcode mnemonics and operand syntax are strictly PIC18 assembly language. Additional mnemonics and assembler directives are documented in this section.

4.3.1 Assembler Format Deviations

The HI-TECH PICC-18 assembler uses a slightly modified form of assembly language to that used in C18 and MPASM.

The HI-TECH PICC-18 assembler uses the operands "`,w`" and "`,f`" to specify the destination register. The W register is selected as the destination when using the "`,w`" operand, and the file register is selected when using the "`,f`" operand or if no destination operand is specified. The case of the letter in the destination operand in not important. The operands "`,0`" or "`,1`" cannot be used to specify the destination.

The PICC-18 assembler also uses the operands "`,b`" and "`,c`" to indicate that a file register is *banked* or *common*. A common register is one that resides in the access bank. Instructions using this operand will have the *RAM access bit* in the instruction cleared by the assembler. A banked register does not reside in the access bank. Instructions using this operand will have the *RAM access bit* in the instruction set by the assembler. The BSR register must be correctly loaded prior to executing a banked instruction to select the appropriate bank. Identifiers that do not use either of these operands are assumed to be banked. A symbol can also be preceded by the characters "`c:`" to indicate that it resides in common memory.

An access bank indicator, such as "`,c`" or "`c:`" is not required when an address used in an instruction is absolute and the value of the address is within the access bank. The assembler will determine from the address that this is the case. However, these indicators must be used with all unresolved identifiers. For example, the following instructions show the WREG first being moved to an absolute location and then to an address represented by an identifier. The op codes for these instructions, assuming that the address assigned to `_foo` is 0516h, are shown.

```
6EE5 movwf 0FE5h
6E16 movwf _foo,c
6F16 movwf _foo,b
6F16 movwf _foo
```

Notice that first two instruction have the *RAM access bit* (bit 8 of the op-code) cleared, but that it is set in the last two instructions.

The `retfie` instruction may be followed by "`f`" (no comma) to indicate that the shadow registers should be retrieved and copied to their corresponding registers on execution.

The compiler also supports the use of the PIC10/12/16 compiler pseudo instructions `LJMP` and `FCALL`. These map to a regular `GOTO` and `CALL` PIC18 instruction, respectively. This support allows for easier porting of assembly code from PIC10/12/16 devices to the PIC18 architecture. Avoid using these pseudo instructions in projects developed for PIC18 devices.

4.3.2 Pre-defined Macros

The file `sfr.h`, contained in the SOURCES directory contains useful definitions for assembler programming. In particular it contains an assembler macro called `loadfsr`, which can be used when you require any of the FSR registers to be loaded. The two arguments to this macro are the FSR register number and the value to be loaded. For example:

```
loadfsr 2,1FFh
```

which will load FSR2 with the value 1FFh. This macro should be used in preference to the `lfsr` instruction.

The `BANKMASK` macro can also be used to mask an address for using in a file register instruction. On all PIC18 devices, it performs a bitwise AND operation with the value 0xFF. Do not use this macro with operands that should represent a full banked address, for example with the `MOVFF` instruction.

4.3.3 Statement Formats

Legal statement formats are shown in Table 4.2.

The *`label`* field is optional and, if present, should contain one identifier. A label may appear on a line of its own, or precede a mnemonic as shown in the second format.

The third format is only legal with certain assembler directives, such as `MACRO`, `SET` and `EQU`. The *`name`* field is mandatory and should also contain one identifier.

If the assembly file is first processed by the C preprocessor, see Section 2.6.12, then it may also contain lines that form valid preprocessor directives. See Section 3.11.2 for more information on the format for these directives.

There is no limitation on what column or part of the line in which any part of the statement should appear.

4.3.4 Characters

The character set used is standard 7 bit ASCII. Alphabetic case is significant for identifiers, but not mnemonics and reserved words. *Tabs* are treated as equivalent to *spaces*.

Table 4.2: ASPIC18 statement formats

Format 1	*label*:			
Format 2	*label*:	*mnemonic*	*operands*	; *comment*
Format 3	*name*	*pseudo-op*	*operands*	; *comment*
Format 4	; *comment only*			
Format 5	<empty line>			

4.3.4.1 Delimiters

All numbers and identifiers must be delimited by *white space*, non-alphanumeric characters or the end of a line.

4.3.4.2 Special Characters

There are a few characters that are special in certain contexts. Within a macro body, the character `&` is used for token concatenation. To use the bitwise `&` operator within a macro body, escape it by using `&&` instead. In a macro argument list, the *angle brackets* < and > are used to quote macro arguments.

4.3.5 Comments

An assembly comment is initiated with a *semicolon* that is not part of a string or character constant.

If the assembly file is first processed by the C preprocessor, see Section 2.6.12, then it may also contain C or C++ style comments using the standard `/*` ... `*/` and `//` syntax.

4.3.5.1 Special Comment Strings

Several comment strings are appended to assembler instructions by the code generator. These are typically used by the assembler optimizer.

The comment string `;volatile` is used to indicate that the memory location being accessed in the commented instruction is associated with a variable that was declared as `volatile` in the C source code. Accesses to this location which appear to be redundant will not be removed by the assembler optimizer if this string is present.

This comment string may also be used in assembler source to achieve the same effect for locations defined and accessed in assembly code.

The comment string `;wreg free` is placed on some CALL instructions. The string indicates that the WREG was not loaded with a function parameter, i.e. it is not in use. If this string is present, optimizations may be made to assembler instructions before the function call which load the WREG redundantly.

Table 4.3: ASPIC18 numbers and bases

Radix	Format
Binary	digits 0 and 1 followed by `B`
Octal	digits 0 to 7 followed by `O`, `Q`, `o` or `q`
Decimal	digits 0 to 9 followed by `D`, `d` or nothing
Hexadecimal	digits 0 to 9, A to F preceded by `0x` or followed by `H` *or* `h`

4.3.6 Constants

4.3.6.1 Numeric Constants

The assembler performs all arithmetic with signed 32-bit precision.

The default radix for all numbers is 10. Other radices may be specified by a trailing base specifier as given in Table 4.3.

Hexadecimal numbers must have a leading digit (e.g. 0ffffh) to differentiate them from identifiers. Hexadecimal digits are accepted in either upper or lower case.

Note that a binary constant must have an upper case `B` following it, as a lower case `b` is used for temporary (numeric) label backward references.

In expressions, real numbers are accepted in the usual format, and are interpreted as IEEE 32-bit format.

4.3.6.2 Character Constants and Strings

A character constant is a single character enclosed in *single quotes* `'`.

Multi-character constants, or strings, are a sequence of characters, not including *carriage return* or *newline* characters, enclosed within matching quotes. Either *single quotes* `'` or *double quotes* `"` maybe used, but the opening and closing quotes must be the same.

4.3.7 Identifiers

Assembly identifiers are user-defined symbols representing memory locations or numbers. A symbol may contain any number of characters drawn from the alphabetics, numerics and the special characters *dollar,* `$`, *question mark,* `?` and *underscore,* `_`.

The first character of an identifier may not be numeric. The case of alphabetics is significant, e.g. `Fred` is not the same symbol as `fred`. Some examples of identifiers are shown here:

```
An_identifier
an_identifier
an_identifier1
```

```
$
?$_12345
```

4.3.7.1 Significance of Identifiers

Users of other assemblers that attempt to implement forms of data typing for identifiers should note that this assembler attaches no significance to any symbol, and places no restrictions or expectations on the usage of a symbol.

The names of *psects* (program sections) and ordinary symbols occupy separate, overlapping name spaces, but other than this, the assembler does not care whether a symbol is used to represent bytes, words or sports cars. No special syntax is needed or provided to define the addresses of bits or any other data type, nor will the assembler issue any warnings if a symbol is used in more than one context. The instruction and addressing mode syntax provide all the information necessary for the assembler to generate correct code.

4.3.7.2 Assembler-Generated Identifiers

Where a LOCAL directive is used in a macro block, the assembler will generate a unique symbol to replace each specified identifier in each expansion of that macro. These unique symbols will have the form `??`*nnnn* where *nnnn* is a 4 digit number. The user should avoid defining symbols with the same form.

4.3.7.3 Location Counter

The current location within the active program section is accessible via the symbol `$`. This symbol expands to the address of the currently executing instruction. Thus:

```
goto $
```

will represent code that will jump to itself and form an endless loop. By using this symbol and an offset, a relative jump destination to be specified.

The address represented by `$` is a word address and thus any offset to this symbol represents a number of instructions. For example:

```
goto $+1
movlw 8
movwf _foo
```

will skip one instruction.

4.3.7.4 Register Symbols

Code in assembly modules may gain access to the special function registers by including pre-defined assembly header files. The appropriate file can be included by add the line:

```
#include <pic18.inc>
```

to the assembler source file. (Assembly header files use the `.inc` extension.) Note that the file must be included using a C pre-processor directive and hence the option to pre-process assembly files must be enabled when compiling, see Section 2.6.12. This header file contains appropriate commands to ensure that the header file specific for the target device is included into the source file.

These header files contain `EQU` declarations for all byte or multi-byte sized registers and `#define` macros for named bits within byte registers.

4.3.7.5 Symbolic Labels

A label is symbolic alias which is assigned a value equal to its offset within the current psect.

A label definition consists of any valid assembly identifier followed by a *colon*, `:`. The definition may appear on a line by itself or be positioned before a statement. Here are two examples of legitimate labels interspersed with assembly code.

```
frank:
            movlw 1
            goto fin
simon44:    clrf _input
```

Here, the label `frank` will ultimately be assigned the address of the `mov` instruction, and `simon44` the address of the `clrf` instruction. Regardless of how they are defined, the assembler list file produced by the assembler will always show labels on a line by themselves.

Labels may be used (and are preferred) in assembly code rather than using an absolute address. Thus they can be used as the target location for jump-type instructions or to load an address into a register.

Like variables, labels have scope. By default, they may be used anywhere in the module in which they are defined. They may be used by code above their definition. To make a label accessible in other modules, use the GLOBAL directive. See Section 4.3.10.1 for more information.

4.3.8 Expressions

The operands to instructions and directives are comprised of expressions. Expressions can be made up of numbers, identifiers, strings and operators.

Table 4.4: ASPIC18 operators

Operator	Purpose	Example
*	Multiplication	`movlw 4*33`
+	Addition	`bra $+1`
-	Subtraction	`DB 5-2`
/	Division	`movlw 100/4`
= or eq	Equality	`IF inp eq 66`
> or gt	Signed greater than	`IF inp > 40`
>= or ge	Signed greater than or equal to	`IF inp ge 66`
< or lt	Signed less than	`IF inp < 40`
<= or le	Signed less than or equal to	`IF inp le 66`
<> or ne	Signed not equal to	`IF inp <> 40`
low	Low byte of operand	`movlw low(inp)`
high	High byte of operand	`movlw high(1008h)`
highword	High 16 bits of operand	`DW highword(inp)`
mod	Modulus	`movlw 77 mod 4`
&	Bitwise AND	`clrf inp&0ffh`
^	Bitwise XOR (exclusive or)	`movlw inp^80`
|	Bitwise OR	`movlw inp|1`
not	Bitwise complement	`movlw not 055h`
< < or shl	Shift left	`DB inp>>8`
> > or shr	Shift right	`movlw inp shr 2`
rol	Rotate left	`DB inp rol 1`
ror	Rotate right	`DB inp ror 1`
float24	24-bit version of real operand	`DW float24(3.3)`
nul	Tests if macro argument is null	

Operators can be unary (one operand, e.g. `not`) or binary (two operands, e.g. `+`). The operators allowable in expressions are listed in Table 4.4. The usual rules governing the syntax of expressions apply.

The operators listed may all be freely combined in both constant and relocatable expressions. The HI-TECH linker permits relocation of complex expressions, so the results of expressions involving relocatable identifiers may not be resolved until link time.

4.3.9 Program Sections

Program sections, or *psects*, are simply a section of code or data. They are a way of grouping together parts of a program (via the psect's name) even though the source code may not be physically adjacent in the source file, or even where spread over several source files.

The concept of a program section is not a HI-TECH-only feature. Often referred to as blocks or segments in other compilers, these grouping of code and data have long used the names `text`, `bss` and `data`.

A psect is identified by a name and has several attributes. The `PSECT` assembler directive is used to define a psect. It takes as arguments a name and an optional comma-separated list of flags. See Section 4.3.10.3 for full information on psect definitions. Chapter 5 has more information on the operation of the linker and on options that can be used to control psect placement in memory.

The assembler associates no significance to the name of a psect and the linker is also not aware of which are compiler-generated or user-defined psects. Unless defined as `abs` (absolute), psects are relocatable.

The following is an example showing some executable instructions being placed in the `text` psect, and some data being placed in the `rbss` psect.

```
PSECT text,class=CODE
adjust:
   goto  clear_fred
increment:
   incf _fred
PSECT bss,class=BANK0,space=1
fred:
   DS 2
PSECT text,class=CODE
clear_fred:
```

```
        clrf _fred
        return
```

Note that even though the two blocks of code in the `text` psect are separated by a block in the `bss` psect, the two `text` psect blocks will be contiguous when loaded by the linker. In other words, the `incf _fred` instruction will be followed by the `clrf` instruction in the final output. The actual location in memory of the `text` and `bss` psects will be determined by the linker.

Code or data that is not explicitly placed into a psect will become part of the default (unnamed) psect.

4.3.10 Assembler Directives

Assembler *directives*, or *pseudo-ops*, are used in a similar way to instruction mnemonics, but either do not generate code, or generate non-executable code, i.e. data bytes. The directives are listed in Table 4.5, and are detailed below.

4.3.10.1 GLOBAL

`GLOBAL` declares a list of symbols which, if defined within the current module, are made public. If the symbols are not defined in the current module, it is a reference to symbols in external modules. Example:

```
    GLOBAL  lab1,lab2,lab3
```

4.3.10.2 END

`END` is optional, but if present should be at the very end of the code defined in the module. It will terminate the assembly process, and not even blank lines should follow this directive.

If an expression is supplied as an argument, that expression will be used to define the entry point (address) of the program. Whether this is of any use will depend on the type of output debug file being generated and the target platform. It is typically most useful for hosted systems, where an application program may not be located at the reset vector.

For example, if `start_label` is defined at the reset vector:

```
    END  start_label
```

Table 4.5: ASPIC18 assembler directives

Directive	Purpose
GLOBAL	Make symbols accessible to other modules or allow reference to other modules' symbols
END	End assembly
PSECT	Declare or resume program section
ORG	Set location counter
EQU	Define symbol value
SET	Define or re-define symbol value
DB	Define constant byte(s)
DW	Define constant word(s)
DS	Reserve storage
DABS	Define absolute storage
IF	Conditional assembly
ELSIF	Alternate conditional assembly
ELSE	Alternate conditional assembly
ENDIF	End conditional assembly
FNCALL	Inform the linker that one function calls another
FNROOT	Inform the linker that a function is the "root" of a call graph
MACRO	Macro definition
ENDM	End macro definition
BANKSEL	Selection bank of specified address
LOCAL	Define local tabs
ALIGN	Align output to the specified boundary
PAGESEL	Generate set/reset instruction to set PCLATH for this page
PROCESSOR	Define the particular chip for which this file is to be assembled.
REPT	Repeat a block of code n times
IRP	Repeat a block of code with a list
IRPC	Repeat a block of code with a character list
SIGNAT	Define function signature

Table 4.6: PSECT flags

Flag	Meaning
abs	Psect is absolute
bit	Psect holds bit objects
class=*name*	Specify class name for psect
delta=*size*	Size of an addressing unit
global	Psect is global (default)
limit=*address*	Upper address limit of psect
local	Psect is not global
ovrld	Psect will overlap same psect in other modules
pure	Psect is to be read-only
reloc=*boundary*	Start psect on specified boundary
size=*max*	Maximum size of psect
space=*area*	Represents area in which psect will reside
with=*psect*	Place psect in the same page as specified psect

4.3.10.3 PSECT

The PSECT directive declares or resumes a program section. It takes as arguments a name and, optionally, a comma-separated list of flags. The allowed flags are listed in Table 4.6, below.

Once a psect has been declared it may be resumed later by another PSECT directive, however the flags need not be repeated.

- abs defines the current psect as being absolute, i.e. it is to start at location 0. This does not mean that this module's contribution to the psect will start at 0, since other modules may contribute to the same psect.

- The bit flag specifies that a psect hold objects that are 1 bit long. Such psects have a scale value of 8 to indicate that there are 8 addressable units to each byte of storage.

- The class flag specifies a class name for this psect. Class names are used to allow local psects to be referred to by a class name at link time, since they cannot be referred to by their own name. Class names are also useful where psects need only be positioned anywhere within a range of addresses rather than at one specific address.

- The delta flag defines the size of an addressing unit. In other words, the number of bytes covered for an increment in the address.

- A psect defined as `global` will be combined with other `global` psects of the same name from other modules at link time. This is the default behaviour for psects, unless the `local` flag is used.
- The `limit` flag specifies a limit on the highest address to which a psect may extend.
- A psect defined as `local` will not be combined with other `local` psects at link time, even if there are others with the same name. Where there are two `local` psects in the one module, they reference the same psect. A `local` psect may not have the same name as any `global` psect, even one in another module.
- A psect defined as `ovrld` will have the contribution from each module overlaid, rather than concatenated at runtime. `ovrld` in combination with `abs` defines a truly absolute psect, i.e. a psect within which any symbols defined are absolute.
- The `pure` flag instructs the linker that this psect will not be modified at runtime and may therefore, for example, be placed in ROM. This flag is of limited usefulness since it depends on the linker and target system enforcing it.
- The `reloc` flag allows specification of a requirement for alignment of the psect on a particular boundary, e.g. `reloc=100h` would specify that this psect must start on an address that is a multiple of 100h.
- The `size` flag allows a maximum size to be specified for the psect, e.g. `size=100h`. This will be checked by the linker after psects have been combined from all modules.
- The `space` flag is used to differentiate areas of memory which have overlapping addresses, but which are distinct. Psects which are positioned in program memory and data memory may have a different `space` value to indicate that the program space address zero, for example, is a different location to the data memory address zero. Devices which use banked RAM data memory typically have the same `space` value as their full addresses (including bank information) are unique.
- The `with` flag allows a psect to be placed in the same page *with* a specified psect. For example `with=text` will specify that this psect should be placed in the same page as the `text` psect.

Some examples of the use of the `PSECT` directive follow:

```
PSECT fred
PSECT bill,size=100h,global
PSECT joh,abs,ovrld,class=CODE,delta=2
```

4.3.10.4 ORG

The `ORG` directive changes the value of the location counter within the current psect. This means that the addresses set with ORG are relative to the base address of the psect, which is not determined until link time.

The much-abused `ORG` directive does *not* necessarily move the location counter to the absolute address you specify as the operand. This directive is rarely needed in programs.

The argument to `ORG` must be either an absolute value, or a value referencing the current psect. In either case the current location counter is set to the value determined by the argument. It is not possible to move the location counter backward. For example:

```
ORG 100h
```

will move the location counter to the beginning of the current psect plus 100h. The actual location will not be known until link time.

In order to use the `ORG` directive to set the location counter to an absolute value, the directive must be used from within an absolute, overlaid psect. For example:

```
PSECT absdata,abs,ovrld
    ORG 50h
```

4.3.10.5 EQU

This pseudo-op defines a symbol and equates its value to an expression. For example

```
thomas EQU 123h
```

The identifier `thomas` will be given the value 123h. `EQU` is legal only when the symbol has not previously been defined. See also Section 4.3.10.6.

4.3.10.6 SET

This pseudo-op is equivalent to `EQU` except that allows a symbol to be re-defined. For example

```
thomas SET 0h
```

4.3.10.7 DB

DB is used to initialize storage as bytes. The argument is a list of expressions, each of which will be assembled into one byte. Each character of the string will be assembled into one memory location. Examples:

```
alabel: DB  'X',1,2,3,4,
```

Note that because the size of an address unit in ROM is 2 bytes, the DB pseudo-op will initialise a word with the upper byte set to zero.

4.3.10.8 DW

DW operates in a similar fashion to DB, except that it assembles expressions into words. Example:

```
DW -1, 3664h, 'A', 3777Q
```

4.3.10.9 DS

This directive reserves, but does not initialize, memory locations. The single argument is the number of bytes to be reserved. Examples:

```
alabel: DS 23    ;Reserve 23 bytes of memory
xlabel: DS 2+3   ;Reserve 5 bytes of memory
```

4.3.10.10 DABS

This directive allows one or more bytes of memory to be reserved at the specified address. The general form of the directive is:

```
DABS memory_space,address,bytes
```

where *memory_space* is a number representing the memory space in which the reservation will take place, *address* is the address at which the reservation will take place, and *bytes* is the number of bytes that is to be reserved. This directive differs to the DS directive in that it does not allocate space at the current location in the current psect, but instead can be used to reserve memory at any location.

The memory space number is the same as the number specified with the space flag option to psects. Devices with a single flat memory space will typically always use 0 as the space value; devices with separate code and data spaces typically use 0 for the code space and 1 for the data space.

The code generator issues a DABS directive for every user-defined absolute C variable, or for variables that have been allocated an address by the code generator.

4.3.10.11 FNCALL

This directive takes the form:

```
FNCALL fun1,fun2
```

FNCALL is usually used in compiler generated code. It tells the linker that function fun1 calls function fun2. This information is used by the linker when performing call graph analysis. If you write assembler code which calls a C function, use the FNCALL directive to ensure that your assembler function is taken into account. For example, if you have an assembler routine called _fred which calls a C routine called foo(), in your assembler code you should write:

```
FNCALL _fred,_foo
```

4.3.10.12 FNROOT

This directive tells the assembler that a function is a root function and thus forms the root of a call graph. It could either be the C main() function or an interrupt function. For example, the C main module produce the directive:

```
FNROOT _main
```

4.3.10.13 IF, ELSIF, ELSE and ENDIF

These directives implement conditional assembly. The argument to IF and ELSIF should be an absolute expression. If it is non-zero, then the code following it up to the next matching ELSE, ELSIF or ENDIF will be assembled. If the expression is zero then the code up to the next matching ELSE or ENDIF will be skipped.

At an ELSE the sense of the conditional compilation will be inverted, while an ENDIF will terminate the conditional assembly block. Example:

```
IF ABC
   goto aardvark
ELSIF DEF
   goto denver
ELSE
   goto grapes
ENDIF
```

In this example, if ABC is non-zero, the first jmp instruction will be assembled but not the second or third. If ABC is zero and DEF is non-zero, the second jmp will be assembled but the first and third will not. If both ABC and DEF are zero, the third jmp will be assembled. Conditional assembly blocks may be nested.

4.3.10.14 MACRO and ENDM

These directives provide for the definition of macros. The `MACRO` directive should be preceded by the macro name and optionally followed by a comma-separated list of formal parameters. When the macro is used, the macro name should be used in the same manner as a machine opcode, followed by a list of arguments to be substituted for the formal parameters.

For example:

```
;macro: storem
;args:  arg1 - the NAME of the source variable
;       arg2 - the literal value to load
;descr: Loads two registers with the value in the variable:
storem   MACRO   arg1,arg2
   movlw &arg2
   movwf &arg1
ENDM
```

When used, this macro will expand to the 2 instructions in the body of the macro, with the formal parameters substituted by the arguments. Thus:

```
storem tempvar,2
```

expands to:

```
movlw 2
movwf tempvar
```

A point to note in the above example: the `&` character is used to permit the concatenation of macro parameters with other text, but is removed in the actual expansion.

A comment may be suppressed within the expansion of a macro (thus saving space in the macro storage) by opening the comment with a double *semicolon*, `;;`.

When invoking a macro, the argument list must be comma-separated. If it is desired to include a *comma* (or other delimiter such as a *space*) in an argument then *angle brackets* < and > may be used to quote the argument. In addition the *exclamation mark*, `!` may be used to quote a single character. The character immediately following the *exclamation mark* will be passed into the macro argument even if it is normally a comment indicator.

If an argument is preceded by a percent sign `%`, that argument will be evaluated as an expression and passed as a decimal number, rather than as a string. This is useful if evaluation of the argument inside the macro body would yield a different result.

The `nul` operator may be used within a macro to test a macro argument, for example:

```
IF nul     arg3  ; argument was not supplied.
    ...
ELSE             ; argument was supplied
    ...
ENDIF
```

By default, the assembly list file will show macro in an unexpanded format, i.e. as the macro was invoked. Expansion of the macro in the listing file can be shown by using the EXPAND assembler control, see Section 4.3.11.3,

4.3.10.15 LOCAL

The LOCAL directive allows unique labels to be defined for each expansion of a given macro. Any symbols listed after the LOCAL directive will have a unique assembler generated symbol substituted for them when the macro is expanded. For example:

```
down MACRO count
   LOCAL more
   more: decfsz count
   goto more
ENDM
```

when expanded will include a unique assembler generated label in place of more. For example:

```
down foobar
```

expands to:

```
??0001 decfsz foobar
       goto ??0001
```

if invoked a second time, the label more would expand to ??0002.

4.3.10.16 ALIGN

The ALIGN directive aligns whatever is following, data storage or code etc., to the specified boundary in the psect in which the directive is found. The boundary is specified by a number following the directive and it specifies a number of bytes. For example, to align output to a 2 byte (even) address within a psect, the following could be used.

```
ALIGN 2
```

Note, however, that what follows will only begin on an even absolute address if the psect begins on an even address. The ALIGN directive can also be used to ensure that a psect's length is a multiple of a certain number. For example, if the above ALIGN directive was placed at the end of a psect, the psect would have a length that was always an even number of bytes long.

4.3.10.17 REPT

The REPT directive temporarily defines an unnamed macro, then expands it a number of times as determined by its argument. For example:

```
REPT 3
addwf fred,w
ENDM
```

will expand to

```
addwf fred,w
addwf fred,w
addwf fred,w
```

4.3.10.18 IRP and IRPC

The IRP and IRPC directives operate similarly to REPT, however instead of repeating the block a fixed number of times, it is repeated once for each member of an argument list. In the case of IRP the list is a conventional macro argument list, in the case or IRPC it is each character in one argument. For each repetition the argument is substituted for one formal parameter.

For example:

```
PSECT idata_0
    IRP number,4865h,6C6Ch,6F00h
    DW number
ENDM
PSECT text0
```

would expand to:

```
PSECT idata_0
    DW 4865h
    DW 6C6Ch
    DW 6F00h
PSECT text0
```

Note that you can use local labels and *angle brackets* in the same manner as with conventional macros.

The `IRPC` directive is similar, except it substitutes one character at a time from a string of non-space characters.

For example:

```
PSECT romdata,class=CODE,delta=2
    IRPC char,ABC
    DB 'char'
ENDM
PSECT text
```

will expand to:

```
PSECT romdata,class=CODE,delta=2
    DB 'A'
    DB 'B'
    DB 'C'
PSECT text
```

4.3.10.19 BANKSEL

This directive can be used to generate code to select the bank of the operand. The operand should be the symbol or literal address of an object that resides in the data memory.

The generated code will use a `MOVLB` instruction.

For example:

```
MOVLW   20
BANKSEL (_foobar)         ; select bank for next file instruction
MOVWF   BANKMASK(_foobar) ; write data and mask address
```

4.3.10.20 PROCESSOR

The output of the assembler may vary depending on the target device. The device name is typically set using the `--CHIP` option to the command-line driver PICC18, see Section 2.6.21, or using the assembler `-P` option, see Table 4.1, but can also be set with this directive, e.g.

```
PROCESSOR 16F877
```

4.3.10.21 SIGNAT

This directive is used to associate a 16-bit signature value with a label. At link time the linker checks that all signatures defined for a particular label are the same and produces an error if they are not. The SIGNAT directive is used by the HI-TECH C compiler to enforce link time checking of C function prototypes and calling conventions.

Use the SIGNAT directive if you want to write assembly language routines which are called from C. For example:

```
SIGNAT _fred,8192
```

will associate the signature value 8192 with the symbol _fred. If a different signature value for _fred is present in any object file, the linker will report an error.

4.3.11 Assembler Controls

Assembler controls may be included in the assembler source to control assembler operation such as listing format. These keywords have no significance anywhere else in the program. The control is invoked by the directive OPT followed by the control name. Some keywords are followed by one or more parameters. For example:

```
OPT EXPAND
```

A list of keywords is given in Table 4.7, and each is described further below. Some controls have shorter forms, which are indicated in the table.

4.3.11.1 ASMOPT_OFF and ASMOPT_ON

The ASMOPT_OFF control disables optimization of the subsequent assembly code up to the next ASMOPT_ON control. These controls only have an effect if the assembler optimizer is enabled, see 2.6.47.

4.3.11.2 COND

Any conditional code will be included in the listing output. See also the NOCOND control in Section 4.3.11.6.

4.3.11.3 EXPAND

When EXPAND is in effect, the code generated by macro expansions will appear in the listing output. See also the NOEXPAND control in Section 4.3.11.7.

Table 4.7: PIC18 assembler controls

Control[1]	Meaning	Format
ASMOPT_ON	Optimizer the following code if the assembler optimizer is enabled	OPT ASMOPT_ON
ASMOPT_OFF	Do not optimize the following code	OPT ASMOPT_OFF
COND | CO	Include conditional code in the listing	OPT COND
EXPAND	Expand macros in the listing output	OPT EXPAND
INCLUDE | IC	Textually include another source file	OPT INCLUDE <*pathname*>
LIST | LI	Define options for listing output	OPT LIST [<*listopt*>, ..., <*listopt*>]
NOCOND | NOCO	Leave conditional code out of the listing	OPT NOCOND
NOEXPAND	Disable macro expansion	OPT NOEXPAND
NOLIST | NOLI	Disable listing output	OPT NOLIST
PAGE	Start a new page in the listing output	OPT PAGE
STACK	Specify the stack depth available for a routine	OPT STACK 10
SUBTITLE	Specify the subtitle of the program	OPT SUBTITLE "<*subtitle*>"
TITLE | TT	Specify the title of the program	OPT TITLE "<*title*>"

Table 4.8: LIST control options

List Option	Default	Description
c=*nnn*	80	Set the page (i.e. column) width.
n=*nnn*	59	Set the page length.
t=*ON*|*OFF*	OFF	Truncate listing output lines. The default wraps lines.
p=<*processor*>	n/a	Set the processor type.
r=<*radix*>	hex	Set the default radix to hex, dec or oct.
x=*ON*|*OFF*	OFF	Turn macro expansion on or off.

4.3.11.4 INCLUDE

This control causes the file specified by *pathname* to be textually included at that point in the assembly file. The INCLUDE control must be the last control keyword on the line, for example:

```
OPT INCLUDE "options.h"
```

The driver does not pass any search paths to the assembler, so if the include file is not located in the working directory, the pathname must specify the exact location.

See also the driver option -P in Section 2.6.12 which forces the C preprocessor to preprocess assembly file, thus allowing use of preprocessor directives, such as #include (see Section 3.11.2).

4.3.11.5 LIST

If the listing was previously turned off using the NOLIST control, the LIST control on its own will turn the listing on.

Alternatively, the LIST control may includes options to control the assembly and the listing. The options are listed in Table 4.8.

See also the NOLIST control in Section 4.3.11.8.

4.3.11.6 NOCOND

Using this control will prevent conditional code from being included in the listing output. See also the COND control in Section 4.3.11.2.

4.3.11.7 NOEXPAND

NOEXPAND disables macro expansion in the listing file. The macro call will be listed instead. See also the EXPAND control in Section 4.3.11.3. Assembly macro are discussed in Section 4.3.10.14.

4.3.11.8 NOLIST

This control turns the listing output off from this point onward. See also the `LIST` control in Section 4.3.11.5.

4.3.11.9 NOXREF

`NOXREF` will disable generation of the *raw* cross reference file. See also the `XREF` control in Section 4.3.11.14.

4.3.11.10 PAGE

`PAGE` causes a new page to be started in the listing output. A *Control-L* (*form feed*) character will also cause a new page when encountered in the source.

4.3.11.11 STACK

The `STACK` control is added by the code generator to indicate to the assembler the available stack level for an assembly routine. It is typically placed after an assembly label. The assembler uses this information if code is being optimized. Procedural abstraction can increase stack usage and can only be employed in a routine if this will not cause the hardware stack to overflow.

Misuse of this control can lead to code failure the stack depth specified is not correct.

4.3.11.12 SUBTITLE

`SUBTITLE` defines a subtitle to appear at the top of every listing page, but under the title. The string should be enclosed in *single* or *double quotes*. See also the `TITLE` control in Section 4.3.11.13.

4.3.11.13 TITLE

This control keyword defines a title to appear at the top of every listing page. The string should be enclosed in *single* or *double quotes*. See also the `SUBTITLE` control in Section 4.3.11.12.

4.3.11.14 XREF

`XREF` is equivalent to the driver command line option `--CR` (see Section 2.6.25). It causes the assembler to produce a raw cross reference file. The utility `CREF` should be used to actually generate the formatted cross-reference listing.

4.4 Assembly List Files

The assembler will produce an assembly list file if instructed. The PICC18 driver option `--ASMLIST` is typically used to request generation of such a file, see Section 2.6.19.

The assembly list file shows the assembly output produced by the compiler for both C and assembly source code. If the assembler optimizers are enabled, the assembly output may be different to assembly source code and so is still useful for assembly programming.

The list file is in a human readable form and cannot take any further part in the compilation sequence. It differs from an assembly output file in that it contains address and op-code data. In addition, the assembler optimizer simplifies some expressions and removes some assembler directives from the listing file for clarity, although these directives are included in the true assembly output files. If you are using the assembly list file to look at the code produced by the compiler, you may wish to turn off the assembler optimizer so that all the compiler-generated directives are shown in this file. Re-enable the optimizer when continuing development. Section 2.6.47 gives more information on controlling the optimizers.

Provided the link stage has successfully concluded, the listing file will be updated by the linker so that it contains absolute addresses and symbol values. Thus you may use the assembler list file to determine the position of, and exact op codes of, instructions.

There is one assembly list file produce by the assembler for each assembly file passed to it, and so there will be one file produced for all the C source code in a project, including p-code based library code. This file will also contains some of the C initialization that forms part of the runtime startup code. There will also be one file produced for each assembly source file. There is typically at least one assembly file in each project, that containing some of the runtime startup file, typically called startup.as.

4.4.1 General Format

The format of the main listing has the form as shown in Section Figure 4.1.

The line numbers purely relate to the assembly list file and are not associated with the lines numbers in the C or assembly source files. Any assembly that begins with a semi- colon indicates it is a comment added by the code generator. Such comments contain either the original source code which corresponds to the generated assembly, or is a comment inserted by the code generator to explain some action taken.

Before the output for each function there is detailed information regarding that function summarized by the code generator. This information relates to register usage, local variable information, functions called and the calling function.

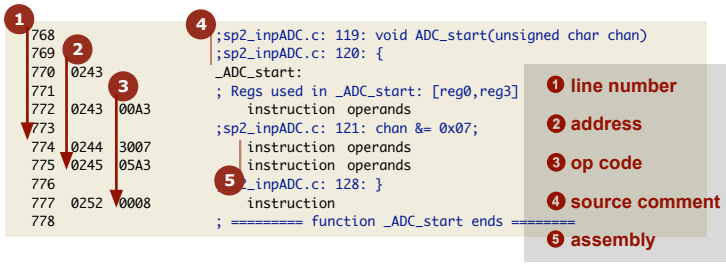


Figure 4.1: General Form of Assembly Listing File

4.4.2 Function Information

For each C function, printed before the function's assembly label (search for the function's name immediately followed by a colon, `:`), is general information relating to the resources used by that function. A typical print out is shown in Figure 4.2. Most of the information is self explanatory, but special comments follow.

The locations shown use the format *`offset[space]`*. For example, a location of 42[BANK0] means that the variables was located in the bank 0 memory space and that it appears at an offset of 42 bytes into the compiled stack component in this space, see Section 3.4.1.1.

Whenever pointer variables are shown, these are often accompanied by the targets the pointer can reference after the arrow `->`, see Section 4.4.3. The `auto` and parameter section of this information is especially useful as the size of pointers is dynamic, see 3.3.12. This information shows the actual number of bytes assigned to each pointer variable.

The tracked objects is generally not used. It indicates the known state of the currently selected RAM bank on entry to the function and at its exit points. It also indicates the bank selection bits that did, or did not, change throughout the function.

The hardware stack information shows how many stack levels were taken up by this function alone and the total levels used by this function and any functions it calls.

Functions which use a non-reentrant model are those which allocate `auto` and parameter variables to a compiled stack and which are, hence, not reentrant.

4.4.3 Pointer Reference Graph

Other important information contained in the assembly list file is the pointer reference graph (look for pointer list with targets: in the list file). This is a list of each and every pointer contained in the program and each target the pointer can reference through the program. The size and type of each target is indicated as well as the size and type of the pointer variable itself.

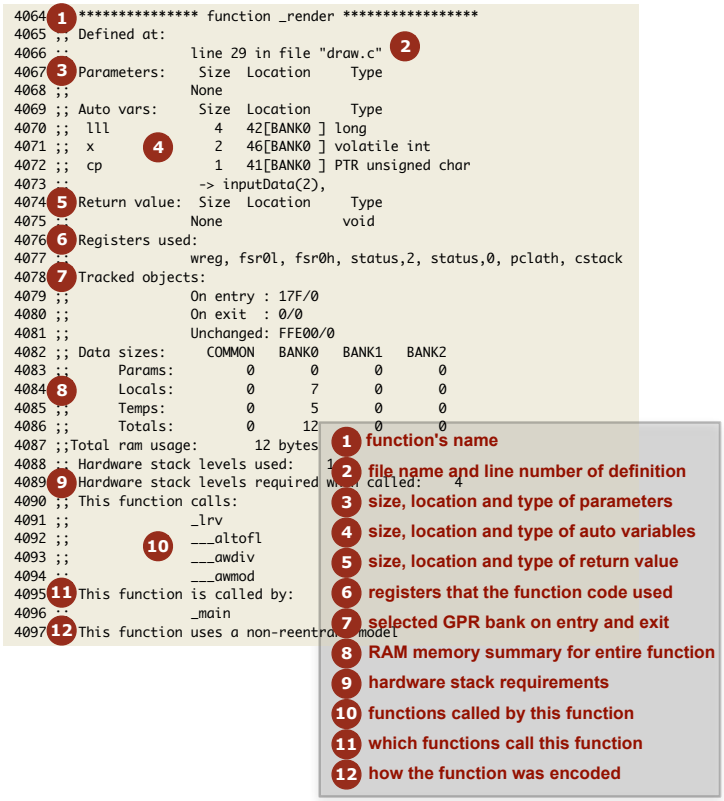

4064 ;; ********** function _render **********
4065 ;; Defined at:
4066 ;; line 29 in file "draw.c"
4067 ;; Parameters: Size Location Type
4068 ;; None
4069 ;; Auto vars: Size Location Type
4070 ;; lll 4 42[BANK0] long
4071 ;; x 2 46[BANK0] volatile int
4072 ;; cp 1 41[BANK0] PTR unsigned char
4073 ;; -> inputData(2),
4074 ;; Return value: Size Location Type
4075 ;; None void
4076 ;; Registers used:
4077 ;; wreg, fsr0l, fsr0h, status,2, status,0, pclath, cstack
4078 ;; Tracked objects:
4079 ;; On entry : 17F/0
4080 ;; On exit : 0/0
4081 ;; Unchanged: FFE00/0
4082 ;; Data sizes: COMMON BANK0 BANK1 BANK2
4083 ;; Params: 0 0 0 0
4084 ;; Locals: 0 7 0 0
4085 ;; Temps: 0 5 0 0
4086 ;; Totals: 0 12 0 0
4087 ;;Total ram usage: 12 bytes
4088 ;; Hardware stack levels used:
4089 ;; Hardware stack levels required when called: 4
4090 ;; This function calls:
4091 ;; _lrv
4092 ;; ___altofl
4093 ;; ___awdiv
4094 ;; ___awmod
4095 ;; This function is called by:
4096 ;; _main
4097 ;; This function uses a non-reentrant model
1 function's name
2 file name and line number of definition
3 size, location and type of parameters
4 size, location and type of auto variables
5 size, location and type of return value
6 registers that the function code used
7 selected GPR bank on entry and exit
8 RAM memory summary for entire function
9 hardware stack requirements
10 functions called by this function
11 which functions call this function
12 how the function was encoded

Figure 4.2: Function Information

For example, the following shows a pointer called `task_tmr` in the C code, and which is local to the function `timer_intr()`. It is a pointer to an `unsigned int` and it is one byte wide. There is only one target to this pointer and it is the member `timer_count` in the structure called `task`. This target variable resides in the `BANK0` class and is two bytes wide.

```
timer_intr@task_tmr PTR unsigned int size(1); Largest target is 2
            -> task.timer_count(BANK0[2]),
```

The pointer reference graph shows both pointers to data objects and pointers to functions.

4.4.4 Call Graph

The other important information block in the assembly list file is the call graph (look for Call Ggraph Tables: in the list file). This is produced for target devices that use a com- piled stack to facilitate local variables, such as function parameters and auto variables. See Section 4.5.4 ?Absolute Variables? for more detailed information on compiled stack operation.

The call graph in the list file shows the information collated and interpreted by the code generator, which is primarily used to allow overlapping of functions? auto-parameter blocks (APBs). The following information can be obtained from studying the call graph.

- The functions in the program that are “root” nodes marking the top of a call tree, and which are called spontaneously
- The functions that the linker deemed were called, or may have been called, during program execution
- The program’s hierarchy of function calls
- The size of the auto and parameter areas within each function’s APB
- The offset of each function’s APB within the compiled stack
- The estimated call tree depth.

These features are discussed below.

A typical call graph may look that shown in Figure 4.3.

The graph starts with the function `main()`. Note that the function name will always be shown in the assembly form, thus the function `main()` appears as the symbol `_main`. `main()` is always a root of a call tree. Interrupt functions will form separate trees.

All the functions that `main()` calls, or may call, are shown below. These have been grouped in the orange box in the figure. A function’s inclusion into the call graph does not imply the function was actually called, but there is a possibility that the function was called. For example, code such as:

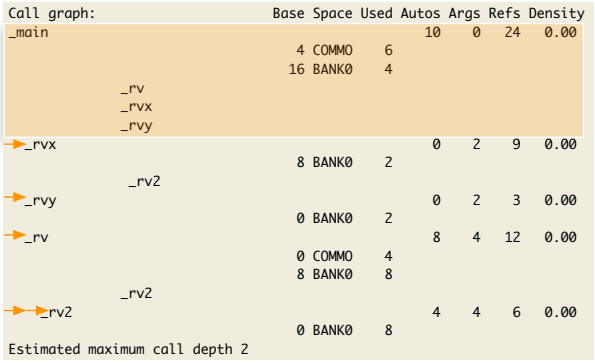


Figure 4.3: Call Graph Form

```
int test(int a) {
   if(a)
      foo();
   else
      bar();
}
```

will list `foo()` and `bar()` under `test()`, as either may be called. If `a` is always true, then the function `bar()` will never be called even though it appears in the call graph.

In addition to these functions there is information relating to the memory allocated in the compiled stack for `main()`. This memory will be used for `auto`, temporary and parameter variables defined in `main()`. The only difference between an `auto` and temporary variable is that `auto` variables are defined by the programmer, and temporaries are defined by the compiler, but both behave in the same way.

In the orange box for `main()` you can see that it defines 10 `auto` and temporary variable. It defines no parameters (`main()` never has parameters). There is a total of 24 references in the assembly code to local objects in `main()`.

Rather than the compiled stack being one memory allocation in one memory space, it can have components placed in multiple memory spaces to utilize all available memory of the target device. This break down is shown under the memory summary line for each function. In this example, it shows that some of the local objects for `main()` are placed in the common memory, but others are placed in bank 0 data RAM.

The *Used* column indicates how many bytes of memory are used by each section of the compiled stack and the *Space* column indicates in which space that has been placed. The *Base* value indicates

the offset that block has in the respective section of the compiled stack. For example, the figure tells us `main()` has 6 bytes of memory allocated at an offset of 4 in the compiled stack section that lives in common memory. It also has 4 bytes of memory allocated in bank 0 memory at an offset of 16 in the bank 0 compiled stack component.

Below the information for `main()` (outside the orange box) you will see the same information repeated for the functions that `main()` called, viz. `rv()`, `rvx()` and `rvy()`. Indentation is used to indicate the maximum depth that function reaches in the call graph. The arrows in the figure highlight this indentation.

After each tree in the call graph, there is an indication of the maximum call (stack) depth that might be realized by that tree. This may be used as a guide to the stack usage of the program. No definitive value can be given for the program's total stack usage for several reasons:

- Certain parts of the call tree may never be reached, reducing that tree's stack usage.
- The contribution of interrupt (or other) trees to the `main()` tree cannot be determined as the point in `main`'s call tree at which the interrupt (or other function invocation) will occur cannot be known;
- The assembler optimizer may have replaced function calls with jumps to functions, reducing that tree's stack usage.
- The assembler's procedural abstraction optimizations may have added in calls to abstracted routines. (Checks are made to ensure this does not exceed the maximum stack depth.)

The code generator also produces a warning if the maximum stack depth appears to have been exceeded. For the above reasons, this warning, too, is intended to be a only a guide to potential stack problems.

4.4.5 Call Graph Critical Paths

Immediately prior to the call graph tables in the list file are the critical paths for memory usage identified in the call graphs. A critical path is printed for each memory space and for each call graph. Look for a line similar to *Critical Paths under _main in BANK0*, which, for this example, indicates the critical path for the `main()` function (the root of one call graph) in bank 0 memory. There will be one call graph for the function `main()` and another for each interrupt function, and each of these will appear for every memory space the device defines.

A critical path here represents the biggest range of APBs stacked together in as a contiguous block. Essentially, it identifies those functions whose APBs are contributing to the program's memory usage in that particular memory space. If you can reduce the memory usage of these functions in the corresponding memory space, then you will affects the program's total memory usage in that memory space.

This information may be presented as follows.

```
3793 ;; Critical Paths under _main in BANK0
3794 ;;
3795 ;;   _main->_foobar
3796 ;;   _foobar->___flsub
3797 ;;   ___flsub->___fladd
```

In this example, it shows that of all the call graph paths starting from the function `main()`, the path `main()` calls `foobar()`, which calls `flsub()`, which calls `fladd()`, is using the largest block of memory in bank 0 RAM. The exact memory usage of each function is shown in the call graph tables.

The memory used by functions that are not in the critical path will overlap entirely with that in the critical path. Reducing the memory usage of these will have no impact on the memory usage of the entire program.

Chapter 5

Linker and Utilities

5.1 Introduction

HI-TECH C incorporates a relocating assembler and linker to permit separate compilation of C source files. This means that a program may be divided into several source files, each of which may be kept to a manageable size for ease of editing and compilation, then each source file may be compiled separately and finally all the object files linked together into a single executable program.

This chapter describes the theory behind and the usage of the linker. Note however that in most instances it will not be necessary to use the linker directly, as the compiler driver will automatically invoke the linker with all necessary arguments. Using the linker directly is not simple, and should be attempted only by those with a sound knowledge of the compiler and linking in general.

If it is absolutely necessary to use the linker directly, the best way to start is to copy the linker arguments constructed by the compiler driver, and modify them as appropriate. This will ensure that the necessary startup module and arguments are present.

Note also that the linker supplied with HI-TECH C is generic to a wide variety of compilers for several different processors. Not all features described in this chapter are applicable to all compilers.

5.2 Relocation and Psects

The fundamental task of the linker is to combine several relocatable object files into one. The object files are said to be *relocatable* since the files have sufficient information in them so that any references to program or data addresses (e.g. the address of a function) within the file may be adjusted according to where the file is ultimately located in memory after the linkage process. Thus the file is said to be relocatable. Relocation may take two basic forms; relocation by name, i.e.

relocation by the ultimate value of a global symbol, or relocation by psect, i.e. relocation by the base address of a particular section of code, for example the section of code containing the actual executable instructions.

5.3 Program Sections

Any object file may contain bytes to be stored in memory in one or more program sections, which will be referred to as *psects*. These psects represent logical groupings of certain types of code bytes in the program. In general the compiler will produce code in three basic types of psects, although there will be several different types of each. The three basic kinds are text psects, containing executable code, data psects, containing initialised data, and bss psects, containing uninitialised but reserved data.

The difference between the data and bss psects may be illustrated by considering two external variables; one is initialised to the value 1, and the other is not initialised. The first will be placed into the data psect, and the second in the bss psect. The bss psect is always cleared to zeros on startup of the program, thus the second variable will be initialised at run time to zero. The first will however occupy space in the program file, and will maintain its initialised value of 1 at startup. It is quite possible to modify the value of a variable in the data psect during execution, however it is better practice not to do so, since this leads to more consistent use of variables, and allows for restartable and ROMable programs.

For more information on the particular psects used in a specific compiler, refer to the appropriate machine-specific chapter.

5.4 Local Psects

Most psects are `global`, i.e. they are referred to by the same name in all modules, and any reference in any module to a `global` psect will refer to the same psect as any other reference. Some psects are `local`, which means that they are local to only one module, and will be considered as separate from any other psect even of the same name in another module. `Local` psects can only be referred to at link time by a class name, which is a name associated with one or more psects via the `PSECT` directive `class=` in assembler code. See Section 4.3.10.3 for more information on `PSECT` options.

5.5 Global Symbols

The linker handles only symbols which have been declared as `GLOBAL` to the assembler. The code generator generates these assembler directives whenever it encounters global C objects. At the C source level, this means all names which have storage class external and which are not declared

as `static`. These symbols may be referred to by modules other than the one in which they are defined. It is the linker's job to match up the definition of a global symbol with the references to it. Other symbols (local symbols) are passed through the linker to the symbol file, but are not otherwise processed by the linker.

5.6 Link and load addresses

The linker deals with two kinds of addresses; *link* and *load* addresses. Generally speaking the link address of a psect is the address by which it will be accessed at run time. The load address, which may or may not be the same as the link address, is the address at which the psect will start within the output file (HEX or binary file etc.). In the case of the 8086 processor, the link address roughly corresponds to the offset within a segment, while the load address corresponds to the physical address of a segment. The segment address is the load address divided by 16.

Other examples of link and load addresses being different are; an initialised data psect that is copied from ROM to RAM at startup, so that it may be modified at run time; a banked text psect that is mapped from a physical (== load) address to a virtual (== link) address at run time.

The exact manner in which link and load addresses are used depends very much on the particular compiler and memory model being used.

5.7 Operation

A command to the linker takes the following form:

```
hlink[1] options files ...
```

Options is zero or more linker options, each of which modifies the behaviour of the linker in some way. *Files* is one or more object files, and zero or more library names. The options recognised by the linker are listed in Table 5.1 and discussed in the following paragraphs.

Table 5.1: Linker command-line options

Option	Effect
`-8`	Use 8086 style segment:offset address form
`-Aclass=low-high,...`	Specify address ranges for a class
`-Cx`	Call graph options (obsolete)
continued...	

[1] In earlier versions of HI-TECH C the linker was called `LINK.EXE`

Table 5.1: Linker command-line options

Option	Effect
-C*psect=class*	Specify a class name for a global psect
-C*baseaddr*	Produce binary output file based at *baseaddr*
-D*class=delta*	Specify a class delta value
-D*symfile*	Produce old-style symbol file
-E*errfile*	Write error messages to *errfile*
-F	Produce `.obj` file with only symbol records
-G*spec*	Specify calculation for segment selectors
-H*symfile*	Generate symbol file
-H+*symfile*	Generate enhanced symbol file
-I	Ignore undefined symbols
-J*num*	Set maximum number of errors before aborting
-K	Prevent overlaying function parameter and auto areas
-L	Preserve relocation items in `.obj` file
-LM	Preserve segment relocation items in `.obj` file
-N	Sort symbol table in map file by address order
-Nc	Sort symbol table in map file by class address order
-Ns	Sort symbol table in map file by space address order
-M*mapfile*	Generate a link map in the named file
-O*outfile*	Specify name of output file
-P*spec*	Specify psect addresses and ordering
-Q*processor*	Specify the processor type (for cosmetic reasons only)
-S	Inhibit listing of symbols in symbol file
-S*class=limit[,bound]*	Specify address limit, and start boundary for a class of psects
-U*symbol*	Pre-enter defined or undefined symbol in table
-V*avmap*	Use file *avmap* to generate an *Avocet* format symbol file
-W*warnlev*	Set warning level (-9 to 9)
-W*width*	Set map file width (>=10)
-X	Remove any local symbols from the symbol file
-Z	Remove trivial local symbols from the symbol file

5.7.1 Numbers in linker options

Several linker options require memory addresses or sizes to be specified. The syntax for all these is similar. By default, the number will be interpreted as a decimal value. To force interpretation as a hex number, a trailing `H` should be added, e.g. `765FH` will be treated as a hex number.

5.7.2 -A*class=low-high,...*

Normally psects are linked according to the information given to a `-P` option (see below) but sometimes it is desired to have a class of psects linked into more than one non-contiguous address range. This option allows a number of address ranges to be specified for a class. For example:

```
-ACODE=1020h-7FFEh,8000h-BFFEh
```

specifies that the class `CODE` is to be linked into the given address ranges. Note that a contribution to a psect from one module cannot be split, but the linker will attempt to pack each block from each module into the address ranges, starting with the first specified.

Where there are a number of identical, contiguous address ranges, they may be specified with a repeat count, e.g.

```
-ACODE=0-FFFFhx16
```

specifies that there are 16 contiguous ranges each 64k bytes in size, starting from zero. Even though the ranges are contiguous, no code will straddle a 64k boundary. The repeat count is specified as the character `x` or `*` after a range, followed by a count.

5.7.3 -C*x*

This option is now obsolete.

5.7.4 -C*psect=class*

This option will allow a psect to be associated with a specific class. Normally this is not required on the command line since classes are specified in object files.

5.7.5 -D*class=delta*

This option allows the *delta* value for psects that are members of the specified class to be defined. The delta value should be a number and represents the number of bytes per addressable unit of objects within the psects. Most psects do not need this option as they are defined with a *delta* value.

5.7.6 -D*symfile*

Use this option to produce an old-style symbol file. An old-style symbol file is an ASCII file, where each line has the link address of the symbol followed by the symbol name.

5.7.7 -E*errfile*

Error messages from the linker are written to standard error (file handle 2). Under DOS there is no convenient way to redirect this to a file (the compiler drivers will redirect standard error if standard output is redirected). This option will make the linker write all error messages to the specified file instead of the screen, which is the default standard error destination.

5.7.8 -F

Normally the linker will produce an object file that contains both program code and data bytes, and symbol information. Sometimes it is desired to produce a symbol-only object file that can be used again in a subsequent linker run to supply symbol values. The `-F` option will suppress data and code bytes from the output file, leaving only the symbol records.

This option can be used when producing more than one hex file for situations where the program is contained in different memory devices located at different addresses. The files for one device are compiled using this linker option to produce a symbol-only object file; this is then linked with the files for the other device. The process can then be repeated for the other files and device.

5.7.9 -G*spec*

When linking programs using segmented, or bank-switched psects, there are two ways the linker can assign segment addresses, or *selectors*, to each segment. A *segment* is defined as a contiguous group of psects where each psect in sequence has both its link and load address concatenated with the previous psect in the group. The segment address or selector for the segment is the value derived when a segment type relocation is processed by the linker.

By default the segment selector will be generated by dividing the base load address of the segment by the relocation quantum of the segment, which is based on the `reloc=` flag value given to psects at the assembler level. This is appropriate for 8086 real mode code, but not for protected mode or some bank-switched arrangements. In this instance the `-G` option is used to specify a method for calculating the segment selector. The argument to `-G` is a string similar to:

```
A/10h-4h
```

where `A` represents the load address of the segment and `/` represents division. This means "Take the load address of the psect, divide by 10 hex, then subtract 4". This form can be modified by substituting `N` for `A`, `*` for `/` (to represent multiplication), and adding rather than subtracting a constant. The token `N` is replaced by the ordinal number of the segment, which is allocated by the linker. For example:

```
N*8+4
```

means "take the segment number, multiply by 8 then add 4". The result is the segment selector. This particular example would allocate segment selectors in the sequence 4, 12, 20, ... for the number of segments defined. This would be appropriate when compiling for 80286 protected mode, where these selectors would represent LDT entries.

5.7.10 -H*symfile*

This option will instruct the linker to generate a symbol file. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is `l.sym`.

5.7.11 -H+*symfile*

This option will instruct the linker to generate an *enhanced* symbol file, which provides, in addition to the standard symbol file, class names associated with each symbol and a segments section which lists each class name and the range of memory it occupies. This format is recommended if the code is to be run in conjunction with a debugger. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is `l.sym`.

5.7.12 -J*errcount*

The linker will stop processing object files after a certain number of errors (other than warnings). The default number is 10, but the `-J` option allows this to be altered.

5.7.13 -K

For compilers that use a compiled stack, the linker will try and overlay function auto and parameter areas in an attempt to reduce the total amount of RAM required. For debugging purposes, this feature can be disabled with this option.

5.7.14 -I

Usually failure to resolve a reference to an undefined symbol is a fatal error. Use of this option will cause undefined symbols to be treated as warnings instead.

5.7.15 -L

When the linker produces an output file it does not usually preserve any relocation information, since the file is now absolute. In some circumstances a further "relocation" of the program will be done at load time, e.g. when running a .exe file under DOS or a `.prg` file under TOS. This requires that some

information about what addresses require relocation is preserved in the object (and subsequently the executable) file. The -L option will generate in the output file one null relocation record for each relocation record in the input.

5.7.16 -LM

Similar to the above option, this preserves relocation records in the output file, but only segment relocations. This is used particularly for generating `.exe` files to run under DOS.

5.7.17 -M*mapfile*

This option causes the linker to generate a link map in the named file, or on the standard output if the file name is omitted. The format of the map file is illustrated in Section 5.9.

5.7.18 -N, -Ns and-Nc

By default the symbol table in the link map will be sorted by name. The `-N` option will cause it to be sorted numerically, based on the value of the symbol. The `-Ns` and `-Nc` options work similarly except that the symbols are grouped by either their *space* value, or class.

5.7.19 -O*outfile*

This option allows specification of an output file name for the linker. The default output file name is `l.obj`. Use of this option will override the default.

5.7.20 -P*spec*

Psects are linked together and assigned addresses based on information supplied to the linker via `-P` options. The argument to the `-P` option consists basically of *comma*-separated sequences thus:

```
-Ppsect=lnkaddr+min/ldaddr+min,psect=lnkaddr/ldaddr, ...
```

There are several variations, but essentially each psect is listed with its desired link and load addresses, and a minimum value. All values may be omitted, in which case a default will apply, depending on previous values.

The minimum value, *min*, is preceded by a + sign, if present. It sets a minimum value for the link or load address. The address will be calculated as described below, but if it is less than the minimum then it will be set equal to the minimum.

The link and load addresses are either numbers as described above, or the names of other psects or classes, or special tokens. If the link address is a negative number, the psect is linked in reverse

order with the top of the psect appearing at the specified address minus one. Psects following a negative address will be placed before the first psect in memory. If a link address is omitted, the psect's link address will be derived from the top of the previous psect, e.g.

```
-Ptext=100h,data,bss
```

In this example the text psect is linked at 100 hex (its load address defaults to the same). The `data` psect will be linked (and loaded) at an address which is 100 hex plus the length of the `text` psect, rounded up as necessary if the data psect has a `reloc=` value associated with it. Similarly, the `bss` psect will concatenate with the `data` psect. Again:

```
-Ptext=-100h,data,bss
```

will link in ascending order `bss`, data then `text` with the top of text appearing at address 0ffh.

If the load address is omitted entirely, it defaults to the same as the link address. If the *slash* / character is supplied, but no address is supplied after it, the load address will concatenate with the previous psect, e.g.

```
-Ptext=0,data=0/,bss
```

will cause both `text` and `data` to have a link address of zero, text will have a load address of 0, and data will have a load address starting after the end of `text`. The bss psect will concatenate with `data` for both link and load addresses.

The load address may be replaced with a *dot* . character. This tells the linker to set the load address of this psect to the same as its link address. The link or load address may also be the name of another (already linked) psect. This will explicitly concatenate the current psect with the previously specified psect, e.g.

```
-Ptext=0,data=8000h/,bss/. -Pnvram=bss,heap
```

This example shows `text` at zero, `data` linked at 8000h but loaded after `text`, `bss` is linked and loaded at 8000h plus the size of `data`, and `nvram` and `heap` are concatenated with `bss`. Note here the use of two `-P` options. Multiple `-P` options are processed in order.

If -A options have been used to specify address ranges for a class then this class name may be used in place of a link or load address, and space will be found in one of the address ranges. For example:

```
-ACODE=8000h-BFFEh,E000h-FFFEh
-Pdata=C000h/CODE
```

This will link `data` at C000h, but find space to load it in the address ranges associated with `CODE`. If no sufficiently large space is available, an error will result. Note that in this case the `data` psect will still be assembled into one contiguous block, whereas other psects in the class `CODE` will be distributed into the address ranges wherever they will fit. This means that if there are two or more psects in class `CODE`, they may be intermixed in the address ranges.

Any psects allocated by a `-P` option will have their load address range subtracted from any address ranges specified with the `-A` option. This allows a range to be specified with the `-A` option without knowing in advance how much of the lower part of the range, for example, will be required for other psects.

5.7.21 -Q*processor*

This option allows a processor type to be specified. This is purely for information placed in the map file. The argument to this option is a string describing the processor.

5.7.22 -S

This option prevents symbol information relating from being included in the symbol file produced by the linker. Segment information is still included.

5.7.23 -S*class=limit[, bound]*

A class of psects may have an upper address *limit* associated with it. The following example places a limit on the maximum address of the `CODE` class of psects to one less than 400h.

```
-SCODE=400h
```

Note that to set an upper limit to a psect, this must be set in assembler code (with a `limit=` flag on a `PSECT` directive).

If the *bound* (boundary) argument is used, the class of psects will start on a multiple of the bound address. This example places the `FARCODE` class of psects at a multiple of 1000h, but with an upper address limit of 6000h:

```
-SFARCODE=6000h,1000h
```

5.7.24 -U*symbol*

This option will enter the specified symbol into the linker's symbol table. The symbol may either be defined or undefined.

Symbols may be defined to be equal to another symbol or a numerical value, e.g.

```
-U_myUndefinedSymbol
-U_myDefinedSymbol=0x55
-U_equatedSymbol=_foobar
```

5.7.25 -V*avmap*

To produce an *Avocet* format symbol file, the linker needs to be given a map file to allow it to map psect names to *Avocet* memory identifiers. The avmap file will normally be supplied with the compiler, or created automatically by the compiler driver as required.

5.7.26 -W*num*

The `-W` option can be used to set the warning level, in the range -9 to 9, or the width of the map file, for values of *num* >= 10.

`-W9` will suppress all warning messages. `-W0` is the default. Setting the warning level to -9 (-W-9) will give the most comprehensive warning messages.

5.7.27 -X

Local symbols can be suppressed from a symbol file with this option. `Global` symbols will always appear in the symbol file.

5.7.28 -Z

Some `local` symbols are compiler generated and not of interest in debugging. This option will suppress from the symbol file all local symbols that have the form of a single alphabetic character, followed by a digit string. The set of letters that can start a trivial symbol is currently "`klfLSu`". The `-Z` option will strip any local symbols starting with one of these letters, and followed by a digit string.

5.8 Invoking the Linker

The linker is called `HLINK`, and normally resides in the `BIN` subdirectory of the compiler installation directory. It may be invoked with no arguments, in which case it will prompt for input from standard input. If the standard input is a file, no prompts will be printed. This manner of invocation is generally useful if the number of arguments to `HLINK` is large. Even if the list of files is too long to fit on one line, continuation lines may be included by leaving a *backslash* \ at the end of the preceding line. In this fashion, `HLINK` commands of almost unlimited length may be issued. For example a link command file called `x.lnk` and containing the following text:

```
-Z -OX.OBJ -MX.MAP \
-Ptext=0,data=0/,bss,nvram=bss/. \
X.OBJ Y.OBJ Z.OBJ C:\HT-Z80\LIB\Z80-SC.LIB
```

may be passed to the linker by one of the following:

```
hlink @x.lnk
hlink < x.lnk
```

5.9 Map Files

The map file contains information relating to the relocation of psects and the addresses assigned to symbols within those psects.

5.9.1 Generation

If compilation is being performed via MPLAB IDE, a map file is generated by default without you having to adjust the compiler options. If you are using the driver from the command line then you'll need to use the `-M` option, see Section 2.6.9.

Map files are produced by the linker. If the compilation process is stopped before the linker is executed, then no map file is produced. The linker will still produce a map file even if it encounters errors, which will allow you to use this file to track down the cause of the errors. However, if the linker ultimately reports `too many errors` then it did not run to completion, and the map file will be either not created or not complete. You can use the `--ERRORS` option on the command line, or as an alternate MPLAB IDE setting, to increase the number of errors before the compiler applications give up. See Section 2.6.32 for more information on this option.

5.9.2 Contents

The sections in the map file, in order of appearance, are as follows:

- The compiler name and version number;
- A copy of the command line used to invoke the linker;
- The version number of the object code in the first file linked;
- The machine type;
- A psect summary sorted by the psect's parent object file;

- A psect summary sorted by the psect's CLASS;
- A segment summary;
- Unused address ranges summary; and
- The symbol table

Portions of an example map file, along with explanatory text, are shown in the following sections.

5.9.2.1 General Information

At the top of the map file is general information relating to the execution of the linker.

When analysing a program, always confirm the compiler version number shown in the map file if you have more than one compiler version installed to ensure the desired compiler is being executed.

The chip selected with the `--CHIP` option should appear after the *Machine type* entry.

The *Object code version* relates to the file format used by relocatable object files produced by the assembler. Unless either the assembler or linker have been updated independently, this should not be of concern.

A typical map file may begin something like the following. This example is valid for the referenced device but might differ to the default options used by another device.

```
HI-TECH Software PICC-18 Compiler PRO Edition #V9.80

Linker command line:

--edf=C:\Program files\HI-TECH Software\picc-18\9.80\dat\en_msgs.txt \
-cs -h+structret.sym -z -Q18F452 -ol.obj -Mstructret.map \
-ACODE=00h-03FFFhx2 -ACONST=00h-07FFFh -ASMALLCONST=0600h-06FFhx122 \
-AMEDIUMCONST=0600h-07FFFh -ACOMRAM=00h-07Fh -AABS1=00h-05FFh \
-ABIGRAM=00h-05FFh -ARAM=080h-0FFh,0100h-01FFhx5 -ABANK0=080h-0FFh
-ABANK1=0100h-01FFh -ABANK2=0200h-02FFh -ABANK3=0300h-03FFh \
-ABANK4=0400h-04FFh -ABANK5=0500h-05FFh -ASFR=0F80h-0FFFh \
-preset_vec=00h,intcode,intcodelo,powerup,init,end_init -pramtop=0600h \
-psmallconst=SMALLCONST -pmediumconst=MEDIUMCONST -pconst=CONST \
-AFARRAM=00h-00h -ACONFIG=0300000h-030000Dh -pconfig=CONFIG \
-AIDLOC=0200000h-0200007h -pidloc=IDLOC -AEEDATA=0F00000h-0F000FFh \
-peeprom_data=EEDATA \
-prdata=COMRAM,nvrram=COMRAM,nvbit=COMRAM,rbss=COMRAM,rbit=COMRAM \
-pfarbss=FARRAM,fardata=FARRAM \
-pintsave_regs=BIGRAM,bigbss=BIGRAM,bigdata=BIGRAM -pbss=RAM \
```

```
-pidata=CODE,irdata=CODE,ibigdata=CODE,ifardata=CODE startup.obj main.obj

Object code version is 3.10

Machine type is 18F452
```

The *Linker command line* shown is the entire list of options and files that were passed to the linker for the build recorded by this map file. Remember, these are linker options and not command-line driver options. The linker options are necessarily complex. Fortunately, they rarely need adjusting from their default settings. They are formed by the command-line driver, PICC18, based on the selected target device and the specified driver options. You can often confirm that driver options were valid by looking at the linker options in the map file. For example, if you ask the driver to reserve an area of memory, you should see a change in the linker options used.

If the default linker options must be changed, this can be done indirectly through the driver using the driver `-L-` option, see Section 2.6.8. If you use this option, always confirm the change appears correctly in the map file.

5.9.2.2 Psect Information listed by Module

The next section in the map file lists those modules that made a contribution to the output, and information regarding the psects these modules defined.

This section is heralded by the line that contains the headings:

```
Name   Link  Load  Length  Selector  Space  Scale
```

Under this on the far left is a list of object files. These object files include both files generated from source modules and those that were extracted from object library files. In the case of those from library files, the name of the library file is printed before the object file list.Note that since the code generator combines all C source files (and p-code libraries), there will only be one object file representing the entire C part of the program. The object file corresponding to the runtime startup code is normally present in this list.

The information in this section of the map file can be used to confirm that a module is making a contribution to the output file and to determine the exact psects that each module defines.

Shown are all the psects (under the *Name* column) that were linked into the program from each object file, and information regarding that psect.

The linker deals with two kinds of addresses: link and load. Generally speaking the link address of a psect is the address by which it will be accessed at run time.

The load address, which is often the same as the link address, is the address at which the psect will start within the output file (HEX or binary file etc.). If a psect is used to hold bits, the load

address is irrelevant and is instead used to hold the link address (in bit units) converted into a byte address.

The *Length* of the psect is shown (in units suitable for that psect).

The *Selector* is less commonly used and is of no concern when compiling for PIC18 devices.

The *Space* field is important as it indicates the memory space in which the psect was placed. For Harvard architecture machines, with separate memory spaces (such as PIC18 devices), this field must be used in conjunction with the address to specify an exact storage location. A space of 0 indicates the program memory, and a space of 1 indicates the data memory. See 4.3.10.3.

The *Scale* of a psect indicates the number of address units per byte — this is left blank if the scale is 1 — and typically this will show 8 for psects that hold bit objects. The *Load* address of psects that hold bits is used to display the link address converted into units of bytes, rather than the load address.

TUTORIAL

INTERPRETING THE PSECT LIST The following appears in a map file.

```
            Name   Link   Load   Length Selector  Space  Scale
ext.obj     text     3A     3A       22       30      0
            bss      4B     4B       10       4B      1
            rbit     50      A        2        0      1      8
```

This indicates that one of the files that the linker processed was called `ext.obj`. (This may have been derived from `ext.c` or `ext.as`.) This object file contained a `text` psect, as well as psects called `bss` and `rbit`. The psect `text` was linked at address 3A and `bss` at address 4B. At first glance, this seems to be a problem given that text is 22 words long, however note that they are in different memory areas, as indicated by the *Space* flag (0 for `text` and 1 for `bss`), and so do not occupy the same memory. The psect `rbit` contains bit objects, as indicated by its *Scale* value (its name is a bit of a giveaway too). Again, at first glance there seems there could be an issue with `rbit` linked over the top of `bss`. Their *Space* flags are the same, but since `rbit` contains bit objects, all the addresses shown are bit addresses, as indicated by the *Scale* value of 8. Note that the *Load* address field of `rbit` psect displays the *Link* address converted to byte units, i.e. 50h/8 => Ah.

5.9.2.3 Psect Information listed by Class

The next section in the map file is the same psect information listed by module, but this time grouped into the psects' class.

This section is heralded by the line that contains the headings:

```
TOTAL   Name  Link  Load  Length
```

Under this are the class names followed by those psects which belong to this class, see 4.3.10.3. These psects are the same as those listed by module in the above section; there is no new information contained in this section.

5.9.2.4 Segment Listing

The class listing in the map file is followed by a listing of segments. A segment is conceptual grouping of contiguous psects, and are used by the linker as an aid in psect placement. There is no segment assembler directive and segments cannot be controlled in any way.

This section is heralded by the line that contains the headings:

```
SEGMENTS   Name  Load  Length  Top  Selector  Space  Class
```

The name of a segment is derived from the psect in the contiguous group with the lowest link address. This can lead to confusion with the psect with the same name. Do not read psect information from this section of the map file.

Typically this section of the map file can be ignored by the user.

5.9.2.5 Unused Address Ranges

The last of the memory summaries Just before the symbol table in the map file is a list of memory which was not allocated by the linker. This memory is thus unused. The linker is aware of any memory allocated by the code generator (for absolute variables), and so this free space is accurate.

This section follows the heading:

```
UNUSED ADDRESS RANGES
```

and is followed by a list of classes and the memory still available in each class. If there is more than one range in a class, each range is printed on a separate line. Any paging boundaries within a class are ignored and are not displayed, but the column *Largest block* shows the largest contiguous free space which takes into account any paging in the memory range. If you are looking to see why psects cannot be placed into memory (e.g. cant-find-space type errors) then this important information to study.

Note that the memory associated with a class can overlap that in others, thus the total free space is not simply the addition of all the unused ranges.

5.9.2.6 Symbol Table

The final section in the map file list global symbols that the program defines. This section has a heading:

```
Symbol Table
```

and is followed by two columns in which the symbols are alphabetically listed. As always with the linker, any C derived symbol is shown with its assembler equivalent symbol name. The symbols listed in this table are:

- Global assembly labels;
- Global `EQU`/`SET` assembler directive labels; and
- Linker-defined symbols.

Assembly symbols are made global via the `GLOBAL` assembler directive, see Section 4.3.10.1 for more information. linker-defined symbols act like `EQU` directives, however they are defined by the linker during the link process, and no definition for them will appear in any source or intermediate file.

Non-`static` C functions, and non-`auto` and non-`static` C variables directly map to assembly labels. The name of the label will be the C identifier with a leading *underscore* character. The linker-defined symbols include symbols used to mark the bounds of psects. See Section 3.12.3. The symbols used to mark the base address of each functions' auto and parameter block are also shown. Although these symbols are used to represent the local autos and parameters of a function, they themselves must be globally accessible to allow each calling function to load their contents. The C auto and parameter variable identifiers are local symbols that only have scope in the function in which they are defined.

Each symbol is shown with the psect in which they are placed, and the address which the symbol has been assigned. There is no information encoded into a symbol to indicate whether it represents code or variables, nor in which memory space it resides.

If the psect of a symbol is shown as `(abs)`, this implies that the symbol is not directly associated with a psect as is the case with absolute C variables. Linker-defined symbols showing this as the psect name may be symbols that have never been used throughout the program, or relate to symbols that are not directly associated with a psect.

Note that a symbol table is also shown in each assembler list file. (See Section 2.6.19 for information on generating these files.) These differ to that shown in the map file in that they list all symbols, whether they be of global or local scope, and they only list the symbols used in the module(s) associated with that list file.

5.10 Librarian

The librarian program, LIBR, has the function of combining several object files into a single file known as a library. The purposes of combining several such object modules are several.

- fewer files to link
- faster access
- uses less disk space

In order to make the library concept useful, it is necessary for the linker to treat modules in a library differently from object files. If an object file is specified to the linker, it will be linked into the final linked module. A module in a library, however, will only be linked in if it defines one or more symbols previously known, but not defined, to the linker. Thus modules in a library will be linked only if required. Since the choice of modules to link is made on the first pass of the linker, and the library is searched in a linear fashion, it is possible to order the modules in a library to produce special effects when linking. More will be said about this later.

5.10.1 The Library Format

The modules in a library are basically just concatenated, but at the beginning of a library is maintained a directory of the modules and symbols in the library. Since this directory is smaller than the sum of the modules, the linker can perform faster searches since it need read only the directory, and not all the modules, on the first pass. On the second pass it need read only those modules which are required, seeking over the others. This all minimises disk I/O when linking.

It should be noted that the library format is geared exclusively toward object modules, and is not a general purpose archiving mechanism as is used by some other compiler systems. This has the advantage that the format may be optimized toward speeding up the linkage process.

5.10.2 Using the Librarian

The librarian program is called LIBR, and the format of commands to it is as follows:

```
LIBR options k file.lib file.obj ...
```

Interpreting this, LIBR is the name of the program, *options* is zero or more librarian options which affect the output of the program. *k* is a key letter denoting the function requested of the librarian (replacing, extracting or deleting modules, listing modules or symbols), *file.lib* is the name of the library file to be operated on, and *file.obj* is zero or more object file names.

The librarian options are listed in Table 5.2.

Table 5.2: Librarian command-line options

Option	Effect
`-P`*`width`*	specify page width
`-W`	Suppress non-fatal errors

Table 5.3: Librarian key letter commands

Key	Meaning
`r`	Replace modules
`d`	Delete modules
`x`	Extract modules
`m`	List modules
`s`	List modules with symbols
`o`	Re-order modules

The key letters are listed in Table 5.3.

When replacing or extracting modules, the *`file.obj`* arguments are the names of the modules to be replaced or extracted. If no such arguments are supplied, all the modules in the library will be replaced or extracted respectively. Adding a file to a library is performed by requesting the librarian to replace it in the library. Since it is not present, the module will be appended to the library. If the `r` key is used and the library does not exist, it will be created.

Under the `d` key letter, the named object files will be deleted from the library. In this instance, it is an error not to give any object file names.

The `m` and `s` key letters will list the named modules and, in the case of the `s` keyletter, the symbols defined or referenced within (global symbols only are handled by the librarian). As with the `r` and `x` key letters, an empty list of modules means all the modules in the library.

The `o` key takes a list of module names and re-orders the matching modules in the library file so they have the same order as that listed on the command line. Modules which are not listed are left in their existing order, and will appear after the re-ordered modules.

5.10.3 Examples

Here are some examples of usage of the librarian. The following lists the global symbols in the modules `a.obj`, `b.obj` and `c.obj`:

```
LIBR s file.lib a.obj b.obj c.obj
```

This command deletes the object modules `a.obj`, `b.obj` and `c.obj` from the library file.lib:

```
LIBR d file.lib a.obj b.obj c.obj
```

5.10.4 Supplying Arguments

Since it is often necessary to supply many object file arguments to LIBR, and command lines are restricted to 127 characters by CP/M and MS-DOS, LIBR will accept commands from standard input if no command line arguments are given. If the standard input is attached to the console, LIBR will prompt for input. Multiple line input may be given by using a *backslash* as a continuation character on the end of a line. If standard input is redirected from a file, LIBR will take input from the file, without prompting. For example:

```
libr
libr> r file.lib 1.obj 2.obj 3.obj \
libr> 4.obj 5.obj 6.obj
```

will perform much the same as if the object files had been typed on the command line. The libr> prompts were printed by LIBR itself, the remainder of the text was typed as input.

```
libr <lib.cmd
```

LIBR will read input from lib.cmd, and execute the command found therein. This allows a virtually unlimited length command to be given to LIBR.

5.10.5 Listing Format

A request to LIBR to list module names will simply produce a list of names, one per line, on standard output. The s keyletter will produce the same, with a list of symbols after each module name. Each symbol will be preceded by the letter D or U, representing a definition or reference to the symbol respectively. The -P option may be used to determine the width of the paper for this operation. For example:

```
LIBR -P80 s file.lib
```

will list all modules in file.lib with their global symbols, with the output formatted for an 80 column printer or display.

5.10.6 Ordering of Libraries

The librarian creates libraries with the modules in the order in which they were given on the command line. When updating a library the order of the modules is preserved. Any new modules added to a library after it has been created will be appended to the end.

The ordering of the modules in a library is significant to the linker. If a library contains a module which references a symbol defined in another module in the same library, the module defining the symbol should come after the module referencing the symbol.

5.10.7 Error Messages

LIBR issues various error messages, most of which represent a fatal error, while some represent a harmless occurrence which will nonetheless be reported unless the -W option was used. In this case all warning messages will be suppressed.

5.11 Objtohex

The HI-TECH linker is capable of producing simple binary files, or object files as output. Any other format required must be produced by running the utility program OBJTOHEX. This allows conversion of object files as produced by the linker into a variety of different formats, including various hex formats. The program is invoked thus:

```
OBJTOHEX options inputfile outputfile
```

All of the arguments are optional. If *outputfile* is omitted it defaults to l.hex or l.bin depending on whether the -b option is used. The *inputfile* defaults to l.obj.

The options for OBJTOHEX are listed in Table 5.4. Where an address is required, the format is the same as for HLINK.

5.11.1 Checksum Specifications

If you are generating a HEX file output, please refer to the hexmate section 5.14 for calculating checksums. For OBJTOHEX, the checksum specification allows automated checksum calculation and takes the form of several lines, each line describing one checksum. The syntax of a checksum line is:

```
addr1-addr2 where1-where2 +offset
```

All of *addr1*, *addr2*, *where1*, *where2* and *offset* are hex numbers, without the usual H suffix. Such a specification says that the bytes at *addr1* through to *addr2* inclusive should be summed and the sum placed in the locations *where1* through *where2* inclusive. For an 8 bit checksum these two addresses should be the same. For a checksum stored low byte first, where1 should be less than where2, and vice versa. The *+offset* is optional, but if supplied, the value offset will be used to initialise the checksum. Otherwise it is initialised to zero. For example:

Table 5.4: OBJTOHEX command-line options

Option	Meaning
-8	Produce a CP/M-86 output file
-A	Produce an ATDOS .atx output file
-B*base*	Produce a binary file with offset of *base*. Default file name is l.obj
-C*ckfile*	Read a list of checksum specifications from *ckfile* or standard input
-D	Produce a COD file
-E	Produce an MS-DOS .exe file
-F*fill*	Fill unused memory with words of value *fill* - default value is 0FFh
-I	Produce an *Intel* HEX file with linear addressed extended records.
-L	Pass relocation information into the output file (used with .exe files)
-M	Produce a *Motorola* HEX file (S19, S28 or S37 format)
-N	Produce an output file for Minix
-P*stk*	Produce an output file for an *Atari* ST, with optional stack size
-R	Include relocation information in the output file
-S*file*	Write a symbol file into *file*
-T	Produce a *Tektronix* HEX file.
-TE	Produce an extended TekHEX file.
-U	Produce a COFF output file
-UB	Produce a UBROF format file
-V	Reverse the order of words and long words in the output file
-*n*,*m*	Format either Motorola or Intel HEX file, where *n* is the maximum number of bytes per record and *m* specifies the record size rounding. Non-rounded records are zero padded to a multiple of *m*. *m* itself must be a multiple of 2.

Table 5.5: CREF command-line options

Option	Meaning
-F*prefix*	Exclude symbols from files with a pathname or filename starting with *prefix*
-H*heading*	Specify a heading for the listing file
-L*len*	Specify the page length for the listing file
-O*outfile*	Specify the name of the listing file
-P*width*	Set the listing width
-S*stoplist*	Read file *stoplist* and ignore any symbols listed.
-X*prefix*	Exclude and symbols starting with *prefix*

```
0005-1FFF 3-4 +1FFF
```

This will sum the bytes in 5 through 1FFFH inclusive, then add 1FFFH to the sum. The 16 bit checksum will be placed in locations 3 and 4, low byte in 3. The checksum is initialised with 1FFFH to provide protection against an all zero ROM, or a ROM misplaced in memory. A run time check of this checksum would add the last address of the ROM being checksummed into the checksum. For the ROM in question, this should be 1FFFH. The initialization value may, however, be used in any desired fashion.

5.12 Cref

The cross reference list utility CREF is used to format raw cross-reference information produced by the compiler or the assembler into a sorted listing. A raw cross-reference file is produced with the --CR option to the compiler. The assembler will generate a raw cross-reference file with a -C option (most assemblers) or by using an OPT CRE directive (6800 series assemblers) or a XREF control line (PIC assembler). The general form of the CREF command is:

```
cref options files
```

where *options* is zero or more options as described below and *files* is one or more raw cross-reference files. CREF takes the options listed in Table 5.5.

Each option is described in more detail in the following paragraphs.

5.12.1 -F*prefix*

It is often desired to exclude from the cross-reference listing any symbols defined in a system header file, e.g. <stdio.h>. The -F option allows specification of a path name prefix that will be used to

exclude any symbols defined in a file whose path name begins with that prefix. For example, `-F\` will exclude any symbols from all files with a path name starting with `\`.

5.12.2 -H*heading*

The `-H` option takes a string as an argument which will be used as a header in the listing. The default heading is the name of the first raw cross-ref information file specified.

5.12.3 -L*len*

Specify the length of the paper on which the listing is to be produced, e.g. if the listing is to be printed on 55 line paper you would use a `-L55` option. The default is 66 lines.

5.12.4 -O*outfile*

Allows specification of the output file name. By default the listing will be written to the standard output and may be redirected in the usual manner. Alternatively *`outfile`* may be specified as the output file name.

5.12.5 -P*width*

This option allows the specification of the width to which the listing is to be formatted, e.g. `-P132` will format the listing for a 132 column printer. The default is 80 columns.

5.12.6 -S*stoplist*

The `-S` option should have as its argument the name of a file containing a list of symbols not to be listed in the cross-reference. Multiple stoplists may be supplied with multiple `-S` options.

5.12.7 -X*prefix*

The `-X` option allows the exclusion of symbols from the listing, based on a prefix given as argument to `-X`. For example if it was desired to exclude all symbols starting with the character sequence `xyz` then the option `-Xxyz` would be used. If a digit appears in the character sequence then this will match any digit in the symbol, e.g. `-XX0` would exclude any symbols starting with the letter `X` followed by a digit.

`CREF` will accept wildcard filenames and I/O redirection. Long command lines may be supplied by invoking `CREF` with no arguments and typing the command line in response to the `cref>` prompt. A *backslash* at the end of the line will be interpreted to mean that more command lines follow.

Table 5.6: CROMWELL format types

Key	Format
cod	*Bytecraft* COD file
coff	COFF file format
elf	ELF/DWARF file
eomf51	Extended OMF-51 format
hitech	HI-TECH Software format
icoff	ICOFF file format
ihex	*Intel* HEX file format
mcoff	Microchip COFF file format
omf51	OMF-51 file format
pe	P&E file format
s19	*Motorola* HEX file format

5.13 Cromwell

The CROMWELL utility converts code and symbol files into different formats. The formats available are shown in Table 5.6.

The general form of the CROMWELL command is:

```
CROMWELL options input_files -okey output_file
```

where *options* can be any of the options shown in Table 5.7. *Output_file* (optional) is the name of the output file. The *input_files* are typically the HEX and SYM file. CROMWELL automatically searches for the SDB files and reads those if they are found. The options are further described in the following paragraphs.

5.13.1 -Pname[,architecture]

The -P options takes a string which is the name of the processor used. CROMWELL may use this in the generation of the output format selected. Note that to produce output in COFF format an additional argument to this option which also specifies the processor architecture is required. Hence for this format the usage of this option must take the form: -Pname,architecture. Table 5.8 enumerates the architectures supported for producing COFF files.

5.13.2 -N

To produce some output file formats (e.g. COFF), Cromwell requires that the names of the program memory space psect classes be provided. The names of the classes are given as a comma separated

Table 5.7: CROMWELL command-line options

Option	Description
-P*name[,architecture]*	Processor name and architecture
-N	Identify code classes
-D	Dump input file
-C	Identify input files only
-F	Fake local symbols as global
-O*key*	Set the output format
-I*key*	Set the input format
-L	List the available formats
-E	Strip file extensions
-B	Specify big-endian byte ordering
-M	Strip underscore character
-V	Verbose mode

Table 5.8: -P option architecture arguments for COFF file output.

Architecture	Description
68K	Motorola 68000 series chips
H8/300	Hitachi 8 bit H8/300 chips
H8/300H	Hitachi 16 bit H8/300H chips
SH	Hitachi 32 bit SuperH RISC chips
PIC12	Microchip base-line PIC chips
PIC14	Microchip mid-range PIC chips
PIC16	Microchip high-end (17Cxxx) PIC chips
PIC18	Microchip PIC18 chips
PIC24	Microchip PIC24F and PIC24H chips
PIC30	Microchip dsPIC30 and dsPIC33 chips

list. For example, in the DSPIC C compiler these classes are typically “CODE” and “NEARCODE”, i.e. `-NCODE,NEARCODE`.

5.13.3 -D

The `-D` option is used to display to the screen details about the named input file in a readable format. The input file can be one of the file types as shown in Table 5.6.

5.13.4 -C

This option will attempt to identify if the specified input files are one of the formats as shown in Table 5.6. If the file is recognised, a confirmation of its type will be displayed.

5.13.5 -F

When generating a COD file, this option can be used to force all local symbols to be represented as global symbols. The may be useful where an emulator cannot read local symbol information from the COD file.

5.13.6 -Okey

This option specifies the format of the output file. The *key* can be any of the types listed in Table 5.6.

5.13.7 -Ikey

This option can be used to specify the default input file format. The *key* can be any of the types listed in Table 5.6.

5.13.8 -L

Use this option to show what file format types are supported. A list similar to that given in Table 5.6 will be shown.

5.13.9 -E

Use this option to tell `CROMWELL` to ignore any filename extensions that were given. The default extension will be used instead.

5.13.10 -B

In formats that support different endian types, use this option to specify big-endian byte ordering.

5.13.11 -M

When generating COD files this option will remove the preceding *underscore* character from symbols.

5.13.12 -V

Turns on verbose mode which will display information about operations `CROMWELL` is performing.

5.14 Hexmate

The Hexmate utility is a program designed to manipulate Intel HEX files. Hexmate is a post-link stage utility that provides the facility to:

- Calculate and store variable-length checksum values
- Fill unused memory locations with known data sequences
- Merge multiple Intel hex files into one output file
- Convert INHX32 files to other INHX formats (e.g. INHX8M)
- Detect specific or partial opcode sequences within a hex file
- Find/replace specific or partial opcode sequences
- Provide a map of addresses used in a hex file
- Change or fix the length of data records in a hex file.
- Validate checksums within Intel hex files.

Typical applications for hexmate might include:

- Merging a bootloader or debug module into a main application at build time
- Calculating a checksum over a range of program memory and storing its value in program memory or EEPROM

- Filling unused memory locations with an instruction to send the PC to a known location if it gets lost.
- Storage of a serial number at a fixed address.
- Storage of a string (e.g. time stamp) at a fixed address.
- Store initial values at a particular memory address (e.g. initialise EEPROM)
- Detecting usage of a buggy/restricted instruction
- Adjusting hex file to meet requirements of particular bootloaders

5.14.1 Hexmate Command Line Options

Some of these hexmate operations may be possible from the compiler's command line driver. However, if hexmate is to be run directly, its usage is:

```
hexmate <file1.hex ... fileN.hex> <options>
```

Where *file1.hex* through to *fileN.hex* are a list of input Intel hex files to merge using hexmate. Additional options can be provided to further customize this process. Table 5.9 lists the command line options that hexmate accepts.

The input parameters to hexmate are now discussed in greater detail. Note that any integral values supplied to the hexmate options should be entered as hexadecimal values without leading `0x` or trailing `h` characters. Note also that any address fields specified in these options are to be entered as byte addresses, unless specified otherwise in the `-ADDRESSING` option.

5.14.1.1 specifications,filename.hex

Intel hex files that can be processed by hexmate should be in either INHX32 or INHX8M format. Additional specifications can be applied to each hex file to put restrictions or conditions on how this file should be processed. If any specifications are used they must precede the filename. The list of specifications will then be separated from the filename by a comma.

A *range restriction* can be applied with the specification `rStart-End`. A range restriction will cause only the address data falling within this range to be used. For example:

```
r100-1FF,myfile.hex
```

will use *myfile.hex* as input, but only process data which is addressed within the range *100h-1FFh* (inclusive) to be read from *myfile.hex*.

An *address shift* can be applied with the specification `sOffset` . If an address shift is used, data read from this hex file will be shifted (by the *Offset*) to a new address when generating the output. The offset can be either positive or negative. For example:

Table 5.9: Hexmate command-line options

Option	Effect
-ADDRESSING	Set address fields in all hexmate options to use word addressing or other
-BREAK	Break continuous data so that a new record begins at a set address
-CK	Calculate and store a checksum value
-FILL	Program unused locations with a known value
-FIND	Search and notify if a particular code sequence is detected
-FIND...,DELETE	Remove the code sequence if it is detected (use with caution)
-FIND...,REPLACE	Replace the code sequence with a new code sequence
-FORMAT	Specify maximum data record length or select INHX variant
-HELP	Show all options or display help message for specific option
-LOGFILE	Save hexmate analysis of output and various results to a file
-O*file*	Specify the name of the output file
-SERIAL	Store a serial number or code sequence at a fixed address
-SIZE	Report the number of bytes of data contained in the resultant hex image.
-STRING	Store an ASCII string at a fixed address
-STRPACK	Store an ASCII string at a fixed address using string packing
-W	Adjust warning sensitivity
+	Prefix to any option to overwrite other data in its address range if necessary

```
r100-1FFs2000,myfile.hex
```

will shift the block of data from 100h-1FFh to the new address range *2100h-21FFh*.

Be careful when shifting sections of executable code. Program code shouldn't be shifted unless it can be guaranteed that no part of the program relies upon the absolute location of this code segment.

5.14.1.2 + Prefix

When the + operator precedes a parameter or input file, the data obtained from that parameter will be forced into the output file and will overwrite other data existing within its address range. For example:

```
+input.hex +-STRING@1000="My string"
```

Ordinarily, hexmate will issue an error if two sources try to store differing data at the same location. Using the + operator informs hexmate that if more than one data source tries to store data to the same address, the one specified with a '+' will take priority.

5.14.1.3 -ADDRESSING

By default, all address parameters in hexmate options expect that values will be entered as byte addresses. In some device architectures the native addressing format may be something other than byte addressing. In these cases it would be much simpler to be able to enter address-components in the device's native format. To facilitate this, the `-ADDRESSING` option is used. This option takes exactly one parameter which configures the number of bytes contained per address location. If for example a device's program memory naturally used a 16-bit (2 byte) word-addressing format, the option `-ADDRESSING=2` will configure hexmate to interpret all command line address fields as word addresses. The affect of this setting is global and all hexmate options will now interpret addresses according to this setting. This option will allow specification of addressing modes from one byte-per-address to four bytes-per-address.

5.14.1.4 -BREAK

This option takes a comma separated list of addresses. If any of these addresses are encountered in the hex file, the current data record will conclude and a new data record will recommence from the nominated address. This can be useful to use new data records to force a distinction between functionally different areas of program space. Some hex file readers depend on this.

5.14.1.5 -CK

The -CK option is for calculating a checksum. The usage of this option is:

```
-CK=start-end@destination[+offset][wWidth][tCode][gAlgorithm]
```

where:

- *Start* and *End* specify the address range that the checksum will be calculated over.
- *Destination* is the address where to store the checksum result. This value cannot be within the range of calculation.
- *Offset* is an optional initial value to add to the checksum result. *Width* is optional and specifies the byte-width of the checksum result. Results can be calculated for byte-widths of 1 to 4 bytes. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order.
- *Code* is a hexadecimal code that will trail each byte in the checksum result. This can allow each byte of the checksum result to be embedded within an instruction.
- *Algorithm* is an integer to select which hexmate algorithm to use to calculate the checksum result. A list of selectable algorithms are given in Table 5.10. If unspecified, the default checksum algorithm used is 8 bit addition.

A typical example of the use of the checksum option is:

```
-CK=0-1FFF@2FFE+2100w2
```

This will calculate a checksum over the range `0-1FFFh` and program the checksum result at address `2FFEh`, checksum value will apply an initial offset of `2100h`. The result will be two bytes wide.

5.14.1.6 -FILL

The -FILL option is used for filling unused memory locations with a known value. The usage of this option is:

```
-fill=[const_width:]fill_expr[@address[:end_address]]
```

where:

Table 5.10: Hexmate Checksum Algorithm Selection

Selector	Algorithm description
-4	Subtraction of 32 bit values from initial value
-3	Subtraction of 24 bit values from initial value
-2	Subtraction of 16 bit values from initial value
-1	Subtraction of 8 bit values from initial value
1	Addition of 8 bit values from initial value
2	Addition of 16 bit values from initial value
3	Addition of 24 bit values from initial value
4	Addition of 32 bit values from initial value
7	Fletcher's checksum (8 bit)
8	Fletcher's checksum (16 bit)

- `const_width` has the form `wn` and signifies the width (*n* bytes) of each constant in `fill_expr`. If `const_width` is not specified, the default value is the native width of the architecture. i.e. `--fill=w1:1` with fill every byte with the value 0x01.

- fill_expr can use the syntax (where *const* and *increment* are *n*-byte constants):

  - *const* fill memory with a repeating constant i.e. `--fill=0xBEEF` becomes 0xBEEF, 0xBEEF, 0xBEEF, 0xBEEF
  - *const*`+=`*increment* fill memory with an incrementing constant i.e. `--fill=0xBEEF+=1` becomes 0xBEEF, 0xBEF0, 0xBEF1, 0xBEF2
  - *const*`-=`*increment* fill memory with a decrementing constant i.e. `--fill=0xBEEF-=0x10` becomes 0xBEEF, 0xBEDF, 0xBECF, 0xBEBF
  - *const*`,`*const*`,...,`*const* fill memory with a list of repeating constants i.e. `--fill=0xDEAD,0xBEEF` becomes 0xDEAD,0xBEEF,0xDEAD,0xBEEF

- The options following *fill_expr* result in the following behaviour:

  - `@unused` (or nothing) fill all unused memory with fill_expr i.e. `--fill=0xBEEF@unused` fills all unused memory with 0xBEEF. (This option can only be used with the PICC18 command-line option `--FILL`, see 2.6.33. The driver will expand this to the appropriate ranges and pass these to HEXMATE.)
  - `@`*address* fill a specific address with fill_expr i.e. `--fill=0xBEEF@0x1000` puts 0xBEEF at address 1000h

- `@address:end_address` fill a range of memory with fill_expr i.e. `--fill=0xBEEF@0:0xFF` puts 0xBEEF in unused addresses between 0 and 255

All constants can be expressed in (unsigned) binary, octal, decimal or hexadecimal, as per normal C syntax, so for example 1234 is a decimal value, 0xFF00 is hexadecimal and FF00 is illegal.

5.14.1.7 -FIND

This option is used to detect and log occurrences of an opcode or partial code sequence. The usage of this option is:

`-FIND=Findcode[mMask]@Start-End[/Align][w][t"Title"]`

where:

- *Findcode* is the hexadecimal code sequence to search for and is entered in little endian byte order.
- *Mask* is optional. It allows a bit mask over the Findcode value and is entered in little endian byte order.
- *Start* and *End* limit the address range to search through.
- *Align* is optional. It specifies that a code sequence can only match if it begins on an address which is a multiple of this value. w, if present will cause hexmate to issue a warning whenever the code sequence is detected.
- *Title* is optional. It allows a title to be given to this code sequence. Defining a title will make log-reports and messages more descriptive and more readable. A title will not affect the actual search results.

TUTORIAL

Let's look at some examples. The option `-FIND=3412@0-7FFF/2w` will detect the code sequence `1234h` when aligned on a `2` (two) byte address boundary, between `0h` and `7FFFh`. `w` indicates that a warning will be issued each time this sequence is found.
Another example, `-FIND=3412M0F00@0-7FFF/2wt"ADDXY"` is same as last example but the code sequence being matched is masked with `000Fh`, so hexmate will search for `123xh`. If a byte-mask is used, is must be of equal byte-width to the opcode it is applied to. Any messaging or reports generated by hexmate will refer to this opcode by the name, *ADDXY* as this was the title defined for this search.

If hexmate is generating a log file, it will contain the results of all searches. -FIND accepts whole bytes of hex data from 1 to 8 bytes in length. Optionally, -FIND can be used in conjunction with ,REPLACE or ,DELETE (as described below).

5.14.1.8 -FIND...,DELETE

If DELETE is used in conjunction with a -FIND option and a sequence is found that matches the -FIND criteria, it will be removed. This function should be used with extreme caution and is not recommended for removal of executable code.

5.14.1.9 -FIND...,REPLACE

REPLACE Can only be used in conjunction with a -FIND option. Code sequences that matched the -FIND criteria can be replaced or partially replaced with new codes. The usage for this sub-option is:

```
-FIND...,REPLACE=Code[mMask]
```

where:

- *Code* is a little endian hexadecimal code to replace the sequences that match the -FIND criteria.
- *Mask* is an optional bit mask to specify which bits within *Code* will replace the code sequence that has been matched. This may be useful if, for example, it is only necessary to modify 4 bits within a 16-bit instruction. The remaining 12 bits can masked and be left unchanged.

5.14.1.10 -FORMAT

The -FORMAT option can be used to specify a particular variant of INHX format or adjust maximum record length. The usage of this option is:

```
-FORMAT=Type[,Length]
```

where:

- *Type* specifies a particular INHX format to generate.
- *Length* is optional and sets the maximum number of bytes per data record. A valid length is between 1 and 128, with 16 being the default.

Table 5.11: INHX types used in -FORMAT option

Type	Description
INHX8M	Cannot program addresses beyond 64K.
INHX32	Can program addresses beyond 64K with extended linear address records.
INHX032	INHX32 with initialization of upper address to zero.

TUTORIAL

Consider this case. A bootloader trying to download an INHX32 file fails because it cannot process the extended address records which are part of the INHX32 standard. You know that this bootloader can only program data addressed within the range 0 to 64k, and that any data in the hex file outside of this range can be safely disregarded. In this case, by generating the hex file in INHX8M format the operation might succeed. The hexmate option to do this would be -FORMAT=INHX8M.
Now consider this. What if the same bootloader also required every data record to contain eight bytes of data, no more, no less? This is possible by combining -FORMAT with -FILL. Appropriate use of -FILL can ensure that there are no gaps in the data for the address range being programmed. This will satisfy the minimum data length requirement. To set the maximum length of data records to eight bytes, just modify the previous option to become -FORMAT=INHX8M,8.

The possible types that are supported by this option are listed in Table 5.11. Note that INHX032 is not an actual INHX format. Selection of this type generates an INHX32 file but will also initialize the upper address information to zero. This is a requirement of some device programmers.

5.14.1.11 -HELP

Using -HELP will list all hexmate options. By entering another hexmate option as a parameter of -HELP will show a detailed help message for the given option. For example:

```
-HELP=string
```

will show additional help for the -STRING hexmate option.

5.14.1.12 -LOGFILE

The -LOGFILE option saves hex file statistics to the named file. For example:

```
-LOGFILE=output.log
```

will analyse the hex file that hexmate is generating and save a report to a file named *output.log*.

5.14.1.13 -MASK

Use this option to logically AND a memory range with a particular bitmask. This is used to ensure that the unimplemented bits in program words (if any) are left blank. The usage of this option is as follows:

```
-MASK=hexcode@start-end
```

Where `hexcode` is a hexadecimal value that will be ANDed with data within the `start-end` address range. Multibyte mask values can be entered in little endian byte order.

5.14.1.14 -O*file*

The generated Intel hex output will be created in this file. For example:

```
-Oprogram.hex
```

will save the resultant output to *program.hex*. The output file can take the same name as one of its input files, but by doing so, it will replace the input file entirely.

5.14.1.15 -SERIAL

This option will store a particular hex value at a fixed address. The usage of this option is:

```
-SERIAL=Code[+/-Increment]@Address[+/-Interval][rRepetitions]
```

where:

- *Code* is a hexadecimal value to store and is entered in little endian byte order.
- *Increment* is optional and allows the value of *Code* to change by this value with each repetition (if requested).
- *Address* is the location to store this code, or the first repetition thereof.
- *Interval* is optional and specifies the address shift per repetition of this code.
- *Repetitions* is optional and specifies the number of times to repeat this code.

For example:

```
-SERIAL=000001@EFFE
```

will store hex code `00001h` to address `EFFEh`.

Another example:

```
-SERIAL=0000+2@1000+10r5
```

will store 5 codes, beginning with value `0000` at address `1000h`. Subsequent codes will appear at address intervals of `+10h` and the code value will change in increments of `+2h`.

5.14.1.16 -SIZE

Using the -SIZE option will report the number of bytes of data within the resultant hex image to standard output. The size will also be recorded in the log file if one has been requested.

5.14.1.17 -STRING

The -STRING option will embed an ASCII string at a fixed address. The usage of this option is:

```
-STRING@Address[tCode]="Text"
```

where:

- *Address* is the location to store this string.
- *Code* is optional and allows a byte sequence to trail each byte in the string. This can allow the bytes of the string to be encoded within an instruction.
- *Text* is the string to convert to ASCII and embed.

For example:

```
-STRING@1000="My favourite string"
```

will store the ASCII data for the string, `My favourite string` (including null terminator) at address `1000h`.

Another example:

```
-STRING@1000t34="My favourite string"
```

will store the same string with every byte in the string being trailed with the hex code `34h`.

5.14.1.18 -STRPACK

This option performs the same function as -STRING but with two important differences. Firstly, only the lower seven bits from each character are stored. Pairs of 7 bit characters are then concatenated and stored as a 14 bit word rather than in separate bytes. This is usually only useful for devices where program space is addressed as 14 bit words. The second difference is that -STRING's `t` specifier is not applicable with -STRPACK.

Appendix A

Library Functions

The functions within the standard compiler library are listed in this chapter. Each entry begins with the name of the function. This is followed by information decomposed into the following categories.

Synopsis the C declaration of the function, and the header file in which it is declared.

Description a narrative description of the function and its purpose.

Example an example of the use of the function. It is usually a complete small program that illustrates the function.

Data types any special data types (structures etc.) defined for use with the function. These data types will be defined in the header file named under **Synopsis**.

See also any allied functions.

Return value the type and nature of the return value of the function, if any. Information on error returns is also included

Only those categories which are relevant to each function are used.

__CONFIG

Synopsis

```
#include <htc.h>

__CONFIG(n, data)
```

Description

This macro is used to program the configuration fuses that set the device into various modes of operation.

The macro accepts the number corresponding to the configuration register it is to program, then the 16-Bit value it is to update it with.

16-Bit masks have been defined to describe each programmable attribute available on each device. These attribute masks can be found tabulated in this manual in the Features and Runtime Environment section.

Multiple attributes can be selected by ANDing them together.

Example

```
#include <htc.h>

__CONFIG(1,RC & OSCEN)
__CONFIG(2,WDTPS16 & BORV45)
__CONFIG(4, DEBUGEN)

void
main (void)
{
}
```

See also

__EEPROM_DATA(), __IDLOC()

__EEPROM_DATA

Synopsis

```
#include <htc.h>

__EEPROM_DATA(a,b,c,d,e,f,g,h)
```

Description

This macro is used to store initial values into the device's EEPROM registers at the time of programming.

The macro must be given blocks of 8 bytes to write each time it is called, and can be called repeatedly to store multiple blocks.

__EEPROM_DATA() will begin writing to EEPROM address zero, and will auto-increment the address written to by 8, each time it is used.

Example

```
#include <htc.h>

__EEPROM_DATA(0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07)
__EEPROM_DATA(0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F)

void
main (void)
{
}
```

See also

__CONFIG()

__IDLOC

Synopsis

```
#include <htc.h>

__IDLOC(x)
```

Description

This macro places data into the device's special locations outside of addressable memory reserved for ID. This would be useful for storage of serial numbers etc.

The macro will attempt to write 5 nibbles of data to the 5 locations reserved for ID purposes.

Example

```
#include <htc.h>

__IDLOC(15F01);
/* will store 1, 5, F, 0 and 1 in the ID registers*/

void
main (void)
{
}
```

See also

__EEPROM_DATA(), __CONFIG()

_DELAY()

Synopsis

```
#include <htc.h>

void _delay(unsigned long cycles);
```

Description

This is an inline function that is expanded by the code generator. When called, this routine expands to an inline assembly delay sequence. The sequence will consist of code that delays for the number of cycles that is specified as argument. The argument must be a literal constant.

Example

```
#include <htc.h>

void
main (void)
{
  control |= 0x80;
  _delay(10);    // delay for 10 cycles
  control &= 0x7F;
}
```

See Also

_delay3()

_DELAY3()

Synopsis

```
#include <htc.h>

void _delay3(unsigned char cycles);
```

Description

This is an inline function that is expanded by the code generator. When called, this routine expands to an inline assembly delay sequence. The sequence will consist of code that delays for 3 times the number of cycles that is specified as argument. The argument can be any expression.

Example

```
#include <htc.h>

void
main (void)
{
  control |= 0x80;
  _delay3(10);    // delay for 30 cycles
  control &= 0x7F;
}
```

See Also

_delay()

ABS

Synopsis

```
#include <stdlib.h>

int abs (int j)
```

Description

The **abs()** function returns the absolute value of **j**.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    int a = -5;

    printf("The absolute value of %d is %d\n", a, abs(a));
}
```

See Also

labs(), fabs()

Return Value

The absolute value of **j**.

ACOS

Synopsis

```
#include <math.h>

double acos (double f)
```

Description

The **acos()** function implements the inverse of cos(), i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose cosine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

/* Print acos() values for -1 to 1 in degrees. */

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = acos(i)*180.0/3.141592;
        printf("acos(%f) = %f degrees\n", i, a);
    }
}
```

See Also

sin(), cos(), tan(), asin(), atan(), atan2()

Return Value

An angle in radians, in the range 0 to π

ASCTIME

Synopsis

```
#include <time.h>

char * asctime (struct tm * t)
```

Description

The **asctime()** function takes the time broken down into the **struct tm** structure, pointed to by its argument, and returns a 26 character string describing the current date and time in the format:

Sun Sep 16 01:03:52 1973\n\0

Note the *newline* at the end of the string. The width of each field in the string is fixed. The example gets the current time, converts it to a **struct tm** pointer with localtime(), it then converts this to ASCII and prints it. The time() function will need to be provided by the user (see time() for details).

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("%s", asctime(tp));
}
```

See Also

ctime(), gmtime(), localtime(), time()

Return Value

A pointer to the string.

Note

The example will require the user to provide the time() routine as it cannot be supplied with the compiler.. See time() for more details.

ASIN

Synopsis

```
#include <math.h>

double asin (double f)
```

Description

The **asin()** function implements the converse of sin(), i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose sine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = asin(i)*180.0/3.141592;
        printf("asin(%f) = %f degrees\n", i, a);
    }
}
```

See Also

sin(), cos(), tan(), acos(), atan(), atan2()

Return Value

An angle in radians, in the range - π

ASSERT

Synopsis

```
#include <assert.h>

void assert (int e)
```

Description

This macro is used for debugging purposes; the basic method of usage is to place assertions liberally throughout your code at points where correct operation of the code depends upon certain conditions being true initially. An **assert()** routine may be used to ensure at run time that an assumption holds true. For example, the following statement asserts that the pointer tp is not equal to NULL:

assert(tp);

If at run time the expression evaluates to false, the program will abort with a message identifying the source file and line number of the assertion, and the expression used as an argument to it. A fuller discussion of the uses of **assert()** is impossible in limited space, but it is closely linked to methods of proving program correctness.

Example

```
void
ptrfunc (struct xyz * tp)
{
    assert(tp != 0);
}
```

Note

When required for ROM based systems, the underlying routine _fassert(...) will need to be implemented by the user.

ATAN

Synopsis

```
#include <math.h>

double atan (double x)
```

Description

This function returns the arc tangent of its argument, i.e. it returns an angle e in the range - π

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", atan(1.5));
}
```

See Also

sin(), cos(), tan(), asin(), acos(), atan2()

Return Value

The arc tangent of its argument.

ATAN2

Synopsis

```
#include <math.h>

double atan2 (double x, double x)
```

Description

This function returns the arc tangent of `y/x`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", atan2(10.0, -10.0));
}
```

See Also

sin(), cos(), tan(), asin(), acos(), atan()

Return Value

The arc tangent of `y/x`.

ATOF

Synopsis

```
#include <stdlib.h>

double atof (const char * s)
```

Description

The **atof()** function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a number to a double. The number may be in decimal, normal floating point or scientific notation.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    double i;

    gets(buf);
    i = atof(buf);
    printf("Read %s: converted to %f\n", buf, i);
}
```

See Also

atoi(), atol(), strtod()

Return Value

A double precision floating point number. If no number is found in the string, 0.0 will be returned.

ATOI

Synopsis

```
#include <stdlib.h>

int atoi (const char * s)
```

Description

The **atoi()** function scans the character string passed to it, skipping leading blanks and reading an optional sign. It then converts an ASCII representation of a decimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = atoi(buf);
    printf("Read %s: converted to %d\n", buf, i);
}
```

See Also

xtoi(), atof(), atol()

Return Value

A signed integer. If no number is found in the string, 0 will be returned.

ATOL

Synopsis

```
#include <stdlib.h>

long atol (const char * s)
```

Description

The **atol()** function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a decimal number to a long integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    long i;

    gets(buf);
    i = atol(buf);
    printf("Read %s: converted to %ld\n", buf, i);
}
```

See Also

atoi(), atof()

Return Value

A long integer. If no number is found in the string, 0 will be returned.

BSEARCH

Synopsis

```
#include <stdlib.h>

void * bsearch (const void * key, void * base, size_t n_memb,
size_t size, int (*compar)(const void *, const void *))
```

Description

The **bsearch()** function searches a sorted array for an element matching a particular key. It uses a binary search algorithm, calling the function pointed to by **compar** to compare elements in the array.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct value {
    char name[40];
    int value;
} values[100];

int
val_cmp (const void * p1, const void * p2)
{
    return strcmp(((const struct value *)p1)->name,
                  ((const struct value *)p2)->name);
}

void
main (void)
{
    char inbuf[80];
    int i;
    struct value * vp;
```

```
        i = 0;
        while(gets(inbuf)) {
            sscanf(inbuf,"%s %d", values[i].name, &values[i].value);
            i++;
        }
        qsort(values, i, sizeof values[0], val_cmp);
        vp = bsearch("fred", values, i, sizeof values[0], val_cmp);
        if(!vp)
            printf("Item 'fred' was not found\n");
        else
            printf("Item 'fred' has value %d\n", vp->value);
    }
```

See Also

qsort()

Return Value

A pointer to the matched array element (if there is more than one matching element, any of these may be returned). If no match is found, a null pointer is returned.

Note

The comparison function must have the correct prototype.

CEIL

Synopsis

```
#include <math.h>

double ceil (double f)
```

Description

This routine returns the smallest whole number not less than **f**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double j;

    scanf("%lf", &j);
    printf("The ceiling of %lf is %lf\n", j, ceil(j));
}
```

CGETS

Synopsis

```
#include <conio.h>

char * cgets (char * s)
```

Description

The **cgets()** function will read one line of input from the console into the buffer passed as an argument. It does so by repeated calls to getche(). As characters are read, they are buffered, with *backspace* deleting the previously typed character, and *ctrl-U* deleting the entire line typed so far. Other characters are placed in the buffer, with a *carriage return* or *line feed (newline)* terminating the function. The collected string is null terminated.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type 'exit' to finish\n");
    }
}
```

See Also

getch(), getche(), putch(), cputs()

Return Value

The return value is the character pointer passed as the sole argument.

CLRWDT

Synopsis

```
#include <htc.h>

CLRWDT();
```

Description

This macro is used to clear the device's internal watchdog timer.

Example

```
#include <htc.h>

void
main (void)
{
    WDTCON=1;
      /* enable the WDT */

    CLRWDT();
}
```

CONFIG_READ(), CONFIG_WRITE()

Synopsis

```
#include <htc.h>

unsigned int config_read(void);

void config_write(unsigned char, unsigned int);
```

Description

These functions allow access to the device configuration registers which determine many of the behavioural aspects of the device itself.

config_read() accepts a single parameter to determine which config word will be read. The 16-Bit value contained in the register is returned.

config_write() doesn't return any value. It accepts a second parameter which is a 16-Bit value to be written to the selected register.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned int   value;

    value = config_read(2);  // read register 2
    value |= WDTEN;  // modify value
    config_write(2, value);   // update config register

}
```

See Also

device_id_read(),idloc_read(),idloc_write()

Return Value

config_read() returns the 16-Bit value contained in the nominated configuration register.

Note

The functions **config_read() config_write()** are only applicable to such devices that support this feature.

COS

Synopsis

```
#include <math.h>

double cos (double f)
```

Description

This function yields the cosine of its argument, which is an angle in radians. The cosine is calculated by expansion of a polynomial series approximation.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i*C), cos(i*C));
}
```

See Also

sin(), tan(), asin(), acos(), atan(), atan2()

Return Value

A double in the range -1 to +1.

COSH, SINH, TANH

Synopsis

```
#include <math.h>

double cosh (double f)
double sinh (double f)
double tanh (double f)
```

Description

These functions are the implement hyperbolic equivalents of the trigonometric functions; cos(), sin() and tan().

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", cosh(1.5));
    printf("%f\n", sinh(1.5));
    printf("%f\n", tanh(1.5));
}
```

Return Value

The function **cosh()** returns the hyperbolic cosine value.
The function **sinh()** returns the hyperbolic sine value.
The function **tanh()** returns the hyperbolic tangent value.

CPUTS

Synopsis

```
#include <conio.h>

void cputs (const char * s)
```

Description

The **cputs()** function writes its argument string to the console, outputting *carriage returns* before each *newline* in the string. It calls putch() repeatedly. On a hosted system **cputs()** differs from puts() in that it writes to the console directly, rather than using file I/O. In an embedded system **cputs()** and puts() are equivalent.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type 'exit' to finish\n");
    }
}
```

See Also

cputs(), puts(), putch()

CTIME

Synopsis

```
#include <time.h>

char * ctime (time_t * t)
```

Description

The **ctime()** function converts the time in seconds pointed to by its argument to a string of the same form as described for asctime(). Thus the example program prints the current time and date.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

gmtime(), localtime(), asctime(), time()

Return Value

A pointer to the string.

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

device_id_read()

Synopsis

```
#include <htc.h>

unsigned int device_id_read(void);
```

Description

This function returns the device ID code that is factory-programmed into the chip. This code can be used to identify the device and its revision number.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned int   id_value;
    unsigned int   device_code;
    unsigned char revision_no;

    id_value = device_id_read();
        /* lower 5 bits represent revision number
         * upper 11 bits identify device */
    device_code = (id_value >> 5);
    revision_no = (unsigned char)(id_value & 0x1F);

}
```

See Also

flash_read(), config_read()

Return Value

device_id_read() returns the 16-Bit factory-programmed device id code used to identify the device type and its revision number.

Note

The **device_id_read()** is applicable only to those devices which are capable of reading their own program memory.

DI, EI

Synopsis

```
#include <htc.h>

void ei (void)
void di (void)
```

Description

The **di()** macro disables all interrupts globally (regardless of priority settings), **ei()** re-enables interrupts globally. These are implemented as macros defined in **PIC18.h**. The example shows the use of **ei()** and **di()** around access to a long variable that is modified during an interrupt. If this was not done, it would be possible to return an incorrect value, if the interrupt occurred between accesses to successive words of the count value.

Example

```
#include <htc.h>

long count;

void
interrupt tick (void)
{
    count++;
}

long
getticks (void)
{
    long val;    /* Disable interrupts around access
                    to count, to ensure consistency.*/
    di();
    val = count;
    ei();
    return val;
```

```
}
```

Note

As these macros act on the global interrupt enable bit of the PIC18 processor, ei() will only restore those interrupt sources that were previously enabled.

DIV

Synopsis

```
#include <stdlib.h>

div_t div (int numer, int demon)
```

Description

The **div()** function computes the quotient and remainder of the numerator divided by the denominator.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    div_t x;

    x = div(12345, 66);
    printf("quotient = %d, remainder = %d\n", x.quot, x.rem);
}
```

See Also

udiv(), ldiv(), uldiv()

Return Value

Returns the quotient and remainder into the **div_t** structure.

EEPROM_READ, EEPROM_WRITE

Synopsis

```
#include <htc.h>

unsigned char eeprom_read (unsigned int address);
void eeprom_write (unsigned int address, unsigned char value);
```

Description

These functions allow access to the on-chip eeprom (when present). The eeprom is not in the directly-accessible memory space and a special byte sequence is loaded to the eeprom control registers to access this memory. Writing a value to the eeprom is a slow process and the **eeprom_write()** function polls the appropriate registers to ensure that any previous writes have completed before writing the next datum.

Reading data is completed in the one cycle and no polling is necessary to check for a read completion.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned char data;
    unsigned int address = 0x0010;

    data=eeprom_read(address);
    eeprom_write(address, data);
}
```

See Also

flash_erase, flash_read, flash_write

Note

The high and low priority interrupt are disabled during sensitive sequences required to access EEPROM. Interrupts are restored after the sequence has completed. eeprom_write() will clear the EEIF hardware flag before returning.

Both eeprom_read() and eeprom_write() are available in a similar macro form. The essential difference between the macro and function implementations is that `EEPROM_READ()`, the macro, does not test nor wait for any prior write operations to complete.

EVAL_POLY

Synopsis

```
#include <math.h>

double eval_poly (double x, const double * d, int n)
```

Description

The **eval_poly()** function evaluates a polynomial, whose coefficients are contained in the array **d**, at **x**, for example:

```
y = x*x*d2 + x*d1 + d0.
```

The order of the polynomial is passed in **n**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double x, y;
    double d[3] = {1.1, 3.5, 2.7};

    x = 2.2;
    y = eval_poly(x, d, 2);
    printf("The polynomial evaluated at %f is %f\n", x, y);
}
```

Return Value

A double value, being the polynomial evaluated at **x**.

EXP

Synopsis

```
#include <math.h>

double exp (double f)
```

Description

The **exp()** routine returns the exponential function of its argument, i.e. e to the power of **f**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 0.0 ; f <= 5 ; f += 1.0)
        printf("e to %1.0f = %f\n", f, exp(f));
}
```

See Also

log(), log10(), pow()

FABS

Synopsis

```
#include <math.h>

double fabs (double f)
```

Description

This routine returns the absolute value of its double argument.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f %f\n", fabs(1.5), fabs(-1.5));
}
```

See Also

abs(), labs()

FLASH Routines

Synopsis

```
#include <plib.h>

void EraseFlash(unsigned long startaddr, unsigned long endaddr);
void ReadFlash(unsigned long startaddr, unsigned int num_bytes,
    unsigned  char *flash_array);
void WriteBytesFlash(unsigned long startaddr, unsigned int num_bytes,
    unsigned char *flash_array);
void WriteWordFlash(unsigned long startaddr, unsigned int data);
void WriteBlockFlash(unsigned long startaddr, unsigned char num_blocks,
    unsigned char *flash_array);
```

Description

Flash routines are no long supported in the standard C libraries. Use the flash routines provided by the peripheral libraries whose prototypes are shown above. These are descibed in Section 3.2.7 or in the peripheral library documentation.

FMOD

Synopsis

```
#include <math.h>

double fmod (double x, double y)
```

Description

The function **fmod** returns the remainder of **x/y** as a floating point quantity.

Example

```
#include <math.h>

void
main (void)
{
    double rem, x;

    x = 12.34;
    rem = fmod(x, 2.1);
}
```

Return Value

The floating-point remainder of x/y.

FLOOR

Synopsis

```
#include <math.h>

double floor (double f)
```

Description

This routine returns the largest whole number not greater than **f**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", floor( 1.5 ));
    printf("%f\n", floor( -1.5));
}
```

FREXP

Synopsis

```
#include <math.h>

double frexp (double f, int * p)
```

Description

The **frexp()** function breaks a floating point number into a normalized fraction and an integral power of 2. The integer is stored into the **int** object pointed to by **p**. Its return value x is in the interval (0.5, 1.0) or zero, and **f** equals x times 2 raised to the power stored in ***p**. If **f** is zero, both parts of the result are zero.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;
    int i;

    f = frexp(23456.34, &i);
    printf("23456.34 = %f * 2^%d\n", f, i);
}
```

See Also

ldexp()

GETCH, GETCHE

Synopsis

```
#include <conio.h>

char getch (void)
char getche (void)
```

Description

The **getch()** function reads a single character from the console keyboard and returns it without echoing. The **getche()** function is similar but does echo the character typed.

In an embedded system, the source of characters is defined by the particular routines supplied. By default, the library contains a version of **getch()** that will interface to the Lucifer Debugger. The user should supply an appropriate routine if another source is desired, e.g. a serial port.

The module *getch.c* in the SOURCES directory contains model versions of all the console I/O routines. Other modules may also be supplied, e.g. *ser180.c* has routines for the serial port in a Z180.

Example

```
#include <conio.h>

void
main (void)
{
    char c;

    while((c = getche()) != '\n')
        continue;
}
```

See Also

cgets(), cputs(), ungetch()

GETCHAR

Synopsis

```
#include <stdio.h>

int getchar (void)
```

Description

The **getchar()** routine is a getc(stdin) operation. It is a macro defined in **stdio.h**. Note that under normal circumstances **getchar()** will NOT return unless a *carriage return* has been typed on the console. To get a single character immediately from the console, use the function getch().

Example

```
#include <stdio.h>

void
main (void)
{
    int c;

    while((c = getchar()) != EOF)
        putchar(c);
}
```

See Also

getc(), fgetc(), freopen(), fclose()

Note

This routine is not usable in a ROM based system.

GETS

Synopsis

```
#include <stdio.h>

char * gets (char * s)
```

Description

The **gets()** function reads a line from standard input into the buffer at **s**, deleting the *newline* (cf. fgets()). The buffer is null terminated. In an embedded system, **gets()** is equivalent to cgets(), and results in getche() being called repeatedly to get characters. Editing (with *backspace*) is available.

Example

```
#include <stdio.h>

void
main (void)
{
    char buf[80];

    printf("Type a line: ");
    if(gets(buf))
        puts(buf);
}
```

See Also

fgets(), freopen(), puts()

Return Value

It returns its argument, or NULL on end-of-file.

GMTIME

Synopsis

```
#include <time.h>

struct tm * gmtime (time_t * t)
```

Description

This function converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in **time.h**. The structure is defined in the 'Data Types' section.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = gmtime(&clock);
    printf("It's %d in London\n", tp->tm_year+1900);
}
```

See Also

ctime(), asctime(), time(), localtime()

Return Value

Returns a structure of type **tm**.

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

IDLOC_READ(), IDLOC_WRITE()

Synopsis

```
#include <htc.h>

unsigned char idloc_read(void);

void idloc_write(unsigned char, unsigned char);
```

Description

These functions allow access to the user ID register which can be used to store small amounts of information such as serial numbers, checksums etc.

idloc_read() accepts a single parameter to determine which user ID register to read. The value contained in the register is returned.

idloc_write() doesn't return any value. It accepts a second parameter which is a value to be written to the selected register. Note that only the lower nibble is significant. The upper nibble of the value written will always be 0xF as per Microchip's documentation.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned char   value;

    value = idloc_read(2);  // read register 2
    value++;                // modify value
    idloc_write(2, value);  // update user ID register

}
```

See Also

device_id_read(),config_read(),config_write()

Return Value

idloc_read() returns the value contained in the nominated user ID register.

Note

The functions **idloc_read()** **idloc_write()** are only applicable to such devices that support this feature.

Note also that ICD2 breakpoints should not be set within the **idloc_write()** function. Doing so can result in disrupting the operation of the debugger.

ISALNUM, ISALPHA, ISDIGIT, ISLOWER et. al.

Synopsis

```
#include <ctype.h>

int isalnum (char c)
int isalpha (char c)
int isascii (char c)
int iscntrl (char c)
int isdigit (char c)
int islower (char c)
int isprint (char c)
int isgraph (char c)
int ispunct (char c)
int isspace (char c)
int isupper (char c)
int isxdigit(char c)
```

Description

These macros, defined in **ctype.h**, test the supplied character for membership in one of several overlapping groups of characters. Note that all except **isascii()** are defined for **c**, if **isascii(c)** is true or if c = EOF.

isalnum(c)	c is in 0-9 or a-z or A-Z
isalpha(c)	c is in A-Z or a-z
isascii(c)	c is a 7 bit ascii character
iscntrl(c)	c is a control character
isdigit(c)	c is a decimal digit
islower(c)	c is in a-z
isprint(c)	c is a printing char
isgraph(c)	c is a non-space printable character
ispunct(c)	c is not alphanumeric
isspace(c)	c is a space, tab or newline
isupper(c)	c is in A-Z
isxdigit(c)	c is in 0-9 or a-f or A-F

Example

```
#include <ctype.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = 0;
    while(isalnum(buf[i]))
        i++;
    buf[i] = 0;
    printf("'%s' is the word\n", buf);
}
```

See Also

toupper(), tolower(), toascii()

ISDIG

Synopsis

```
#include <ctype.h>

int isdig (int c)
```

Description

The **isdig**() function tests the input character `c` to see if is a decimal digit (0 – 9) and returns true is this is the case; false otherwise.

Example

```
#include <ctype.h>

void
main (void)
{
    char buf[] = "1998a";
    if(isdig(buf[0]))
        printf("valid type detected\n");
}
```

See Also

isdigit() (listed un isalnum())

Return Value

Zero if the character is a decimal digit; a non-zero value otherwise.

ITOA

Synopsis

```
#include <stdlib.h>

char * itoa (char * buf, int val, int base)
```

Description

The function **itoa** converts the contents of **val** into a string which is stored into **buf**. The conversion is performed according to the radix specified in **base**. **buf** is assumed to reference a buffer which has sufficent space allocated to it.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    itoa(buf, 1234, 16);
    printf("The buffer holds %s\n", buf);
}
```

See Also

strtol(), utoa(), ltoa(), ultoa()

Return Value

This routine returns a copy of the buffer into which the result is written.

LABS

Synopsis

```
#include <stdlib.h>

int labs (long int j)
```

Description

The **labs()** function returns the absolute value of long value **j**.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    long int a = -5;

    printf("The absolute value of %ld is %ld\n", a, labs(a));
}
```

See Also

abs()

Return Value

The absolute value of **j**.

LDEXP

Synopsis

```
#include <math.h>

double ldexp (double f, int i)
```

Description

The **ldexp()** function performs the inverse of frexp() operation; the integer **i** is added to the exponent of the floating point **f** and the resultant returned.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    f = ldexp(1.0, 10);
    printf("1.0 * 2^10 = %f\n", f);
}
```

See Also

frexp()

Return Value

The return value is the integer **i** added to the exponent of the floating point value **f**.

LDIV

Synopsis

```
#include <stdlib.h>

ldiv_t ldiv (long number, long denom)
```

Description

The **ldiv()** routine divides the numerator by the denominator, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer which is less than the absolute value of the mathematical quotient.

The **ldiv()** function is similar to the div() function, the difference being that the arguments and the members of the returned structure are all of type **long int**.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    ldiv_t lt;

    lt = ldiv(1234567, 12345);
    printf("Quotient = %ld, remainder = %ld\n", lt.quot, lt.rem);
}
```

See Also

div(), uldiv(), udiv()

Return Value

Returns a structure of type **ldiv_t**

LOCALTIME

Synopsis

```
#include <time.h>

struct tm * localtime (time_t * t)
```

Description

The **localtime()** function converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in **time.h**. The routine **localtime()** takes into account the contents of the global integer time_zone. This should contain the number of minutes that the local time zone is *westward* of Greenwich. On systems where it is not possible to predetermine this value, **localtime()** will return the same result as **gmtime()**.

Example

```
#include <stdio.h>
#include <time.h>

    char * wday[] = {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
};

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("Today is %s\n", wday[tp->tm_wday]);
}
```

See Also

ctime(), asctime(), time()

Return Value

Returns a structure of type **tm**.

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

LOG, LOG10

Synopsis

```
#include <math.h>

double log (double f)
double log10 (double f)
```

Description

The **log()** function returns the natural logarithm of **f**. The function **log10()** returns the logarithm to base 10 of **f**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("log(%1.0f) = %f\n", f, log(f));
}
```

See Also

exp(), pow()

Return Value

Zero if the argument is negative.

LONGJMP

Synopsis

```
#include <setjmp.h>

void longjmp (jmp_buf buf, int val)
```

Description

The **longjmp()** function, in conjunction with setjmp(), provides a mechanism for non-local goto's. To use this facility, setjmp() should be called with a **jmp_buf** argument in some outer level function. The call from setjmp() will return 0.

To return to this level of execution, **longjmp()** may be called with the same **jmp_buf** argument from an inner level of execution. ***Note*** however that the function which called setjmp() must still be active when **longjmp()** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data. The **val** argument to **longjmp()** will be the value apparently returned from the setjmp(). This should normally be non-zero, to distinguish it from the genuine setjmp() call.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;
```

```
        if(i = setjmp(jb)) {
            printf("setjmp returned %d\n", i);
            exit(0);
        }
        printf("setjmp returned 0 - good\n");
        printf("calling inner...\n");
        inner();
        printf("inner returned - bad!\n");
    }
```

See Also

setjmp()

Return Value

The **longjmp()** routine never returns.

Note

The function which called setjmp() must still be active when **longjmp()** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data.

LTOA

Synopsis

```
#include <stdlib.h>

char * ltoa (char * buf, long val, int base)
```

Description

The function **ltoa** converts the contents of **val** into a string which is stored into **buf**. The conversion is performed according to the radix specified in **base**. **buf** is assumed to reference a buffer which has sufficent space allocated to it.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    ltoa(buf, 12345678L, 16);
    printf("The buffer holds %s\n", buf);
}
```

See Also

strtol(), itoa(), utoa(), ultoa()

Return Value

This routine returns a copy of the buffer into which the result is written.

MEMCMP

Synopsis

```
#include <string.h>

int memcmp (const void * s1, const void * s2, size_t n)
```

Description

The **memcmp()** function compares two blocks of memory, of length **n**, and returns a signed value similar to strncmp(). Unlike strncmp() the comparison does not stop on a null character.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int buf[10], cow[10], i;

    buf[0] = 1;
    buf[2] = 4;
    cow[0] = 1;
    cow[2] = 5;
    buf[1] = 3;
    cow[1] = 3;
    i = memcmp(buf, cow, 3*sizeof(int));
    if(i < 0)
        printf("less than\n");
    else if(i > 0)
        printf("Greater than\n");
    else
        printf("Equal\n");
}
```

See Also

strncpy(), strncmp(), strchr(), memset(), memchr()

Return Value

Returns negative one, zero or one, depending on whether **s1** points to string which is less than, equal to or greater than the string pointed to by **s2** in the collating sequence.

MEMMOVE

Synopsis

```
#include <string.h>

void * memmove (void * s1, const void * s2, size_t n)
```

Description

The **memmove()** function is similar to the function memcpy() except copying of overlapping blocks is handled correctly. That is, it will copy forwards or backwards as appropriate to correctly copy one block to another that overlaps it.

See Also

strncpy(), strncmp(), strchr(), memcpy()

Return Value

The function **memmove()** returns its first argument.

MKTIME

Synopsis

```
#include <time.h>

time_t mktime (struct tm * tmptr)
```

Description

The **mktime()** function converts the local calendar time referenced by the tm structure pointer **tmptr** into a time being the number of seconds passed since Jan 1st 1970, or -1 if the time cannot be represented.

Example

```
#include <time.h>
#include <stdio.h>

void
main (void)
{
    struct tm birthday;

    birthday.tm_year = 75;  // 1975
    birthday.tm_mon = 2;
    birthday.tm_mday = 24;
    birthday.tm_hour = birthday.tm_min = birthday.tm_sec = 0;
    printf("you were born approximately %ld seconds after the unix epoch\n",
 mktime(&birthday));
}
```

See Also

ctime(), asctime()

Return Value

The time contained in the **tm** structure represented as the number of seconds since the 1970 Epoch, or -1 if this time cannot be represented.

MODF

Synopsis

```
#include <math.h>

double modf (double value, double * iptr)
```

Description

The **modf()** function splits the argument **value** into integral and fractional parts, each having the same sign as **value**. For example, -3.17 would be split into the integral part (-3) and the fractional part (-0.17).

The integral part is stored as a double in the object pointed to by **iptr**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i_val, f_val;

    f_val = modf( -3.17, &i_val);
}
```

Return Value

The signed fractional part of **value**.

NOP

Synopsis

```
#include <htc.h>

NOP();
```

Description

Execute NOP instruction here. This is often useful to finetune delays or create a handle for breakpoints. The NOP instruction is sometimes required during some sensitive sequences in hardware.

Example

```
#include <htc.h>

void
crude_delay(unsigned char x) {
   while(x--){
        NOP();  /* Do nothing for 3 cycles */
        NOP();
        NOP();
   }
}
```

OS_TSLEEP

Synopsis

```
#include <task.h>

void os_tsleep(unsigned short tcks)
```

Description

This routine causes the current task to be removed from the run queue for **tcks** clock ticks.

Example

```
#include <task.h>

void
task(void)
{
    while(1) {
        /* sleep for 100 ticks */
        os_tsleep(100);
    }
}
```

POW

Synopsis

```
#include <math.h>

double pow (double f, double p)
```

Description

The **pow()** function raises its first argument, **f**, to the power **p**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("pow(2, %1.0f) = %f\n", f, pow(2, f));
}
```

See Also

log(), log10(), exp()

Return Value

f to the power of **p**.

PRINTF

Synopsis

```
#include <stdio.h>

unsigned int printf (const char * fmt, ...)
```

Description

The **printf()** function is a formatted output routine, operating on stdout. There are corresponding routines operating into a string buffer (**sprintf()**). The **printf()** routine is passed a format string, followed by a list of zero or more arguments. In the format string are conversion specifications, each of which is used to print out one of the argument list values. The **printf()** function performs the text formatting and calls on the **putch()** function to actually send the data to the destination.

Each conversion specification is of the form **%m.nc** where the percent symbol **%** introduces a conversion, followed by an optional width specification **m**. The **n** specification is an optional precision specification (introduced by the dot) and **c** is a letter specifying the type of the conversion.

If the character ***** is used in place of a decimal constant, e.g. in the format **%*d**, then one integer argument will be taken from the list to provide that value. The types of conversion are:

o x X u d

Integer conversion - in radices 8, 16, 16, 10 and 10 respectively. The conversion is signed in the case of **d**, unsigned otherwise. The precision value is the total number of digits to print, and may be used to force leading zeroes. E.g. **%8.4x** will print at least 4 hex digits in an 8 wide field. The letter **X** prints out hexadecimal numbers using the upper case letters *A-F* rather than *a-f* as would be printed when using **x**. When the alternate format is specified, a leading zero will be supplied for the octal format, and a leading 0x or 0X for the hex format.

s

Print a string - the value argument is assumed to be a character pointer. At most **n** characters from the string will be printed, in a field **m** characters wide.

c

The argument is assumed to be a single character and is printed literally.

Any other characters used as conversion specifications will be printed. Thus **%** will produce a single percent sign.

l

Long integer conversion - Preceding the integer conversion key letter with an **l** indicates that the argument list is long.

f

Floating point - **m** is the total width and **n** is the number of digits after the decimal point. If **n** is

omitted it defaults to 6. If the precision is zero, the decimal point will be omitted unless the alternate format is specified.

Example

```
printf("Total = %4d%", 23)
       yields 'Total =   23%'

printf("Size is %lx" , size)
       where size is a long, prints size
       as hexadecimal.

printf("Name = %.8s", "a1234567890")
       yields 'Name = a1234567'

printf("xx%*d", 3, 4)
       yields 'xx  4'

/* vprintf example */

#include <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf("Error: ");
    vprintf(s, ap);
    putchar('\n');
    va_end(ap);
}

void
main (void)
{
    int i;
```

```
        i = 3;
        error("testing 1 2 %d", i);
    }
```

See Also

sprintf()

Return Value

The **printf()** routine returns the number of characters written to stdout.

Note

To use **printf()**, the **putch()** function needs to be defined to output one byte of data to the required destination.

PUTCH

Synopsis

```
#include <conio.h>

void putch (char c)
```

Description

The **putch()** function outputs the character **c** to the console screen, prepending a *carriage return* if the character is a *newline*. In a CP/M or MS-DOS system this will use one of the system I/O calls. In an embedded system this routine, and associated others, will be defined in a hardware dependent way. The standard **putch()** routines in the embedded library interface either to a serial port or to the Lucifer Debugger.

Example

```
#include <conio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putch(*x++);
    putch('\n');
}
```

See Also

cgets(), cputs(), getch(), getche()

PUTCHAR

Synopsis

```
#include <stdio.h>

int putchar (int c)
```

Description

The **putchar()** function is a putc() operation on stdout, defined in **stdio.h**.

Example

```
#include <stdio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putchar(*x++);
    putchar('\n');
}
```

See Also

putc(), getc(), freopen(), fclose()

Return Value

The character passed as argument, or EOF if an error occurred.

Note

This routine is not usable in a ROM based system.

PUTS

Synopsis

```
#include <stdio.h>

int puts (const char * s)
```

Description

The **puts()** function writes the string **s** to the *stdout stream*, appending a *newline*. The null character terminating the string is not copied.

Example

```
#include <stdio.h>

void
main (void)
{
    puts("Hello, world!");
}
```

See Also

fputs(), gets(), freopen(), fclose()

Return Value

EOF is returned on error; zero otherwise.

QSORT

Synopsis

```
#include <stdlib.h>

void qsort (void * base, size_t nel, size_t width,
int (*func)(const void *, const void *))
```

Description

The **qsort()** function is an implementation of the quicksort algorithm. It sorts an array of **nel** items, each of length **width** bytes, located contiguously in memory at **base**. The argument **func** is a pointer to a function used by **qsort()** to compare items. It calls **func** with pointers to two items to be compared. If the first item is considered to be greater than, equal to or less than the second then **func** should return a value greater than zero, equal to zero or less than zero respectively.

Example

```
#include <stdio.h>
#include <stdlib.h>

int aray[] = {
    567, 23, 456, 1024, 17, 567, 66
};

int
sortem (const void * p1, const void * p2)
{
    return *(int *)p1 - *(int *)p2;
}

void
main (void)
{
    register int i;
```

```
    qsort(aray, sizeof aray/sizeof aray[0],
        sizeof aray[0], sortem);
    for(i = 0 ; i != sizeof aray/sizeof aray[0] ; i++)
        printf("%d\t", aray[i]);
    putchar('\n');
}
```

Note

The function parameter must be a pointer to a function of type similar to:

```
int func (const void *, const void *)
```

i.e. it must accept two const void * parameters, and must be prototyped.

RAND

Synopsis

```
#include <stdlib.h>

int rand (void)
```

Description

The **rand()** function is a pseudo-random number generator. It returns an integer in the range 0 to 32767, which changes in a pseudo-random fashion on each call. The algorithm will produce a deterministic sequence if started from the same point. The starting point is set using the **srand()** call. The example shows use of the **time()** function to generate a different starting point for the sequence each time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

See Also

srand()

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

READTIMER*x*

Synopsis

```
#include <htc.h>

READTIMER0();
READTIMER1();
READTIMER3();
```

Description

The macros **READTIMER0()**, **READTIMER1()** and **READTIMER3()** will return the 16-Bit value presently held in the device's corresponding TMR*x*L and TMR*x*H register pair. Use of this macro ensures that the registers are read in the correct order. Timer 2 is an 8-bit timer and does

Example

```
#include <htc.h>

void
main (void)
{
    unsigned int timer1value;

    timer1value = READTIMER1();
}
```

See Also

WRITETIMER*x*()

Return Value

An unsigned integer which is the value held in a 16-Bit timer.

RESET

Synopsis

```
#include <htc.h>

RESET();
```

Description

Execute RESET instruction here.

Example

```
#include <htc.h>

void
test_result(unsigned int error_count) {
   if(error_count != 0){
        printf("An error has been detected - Rebooting...\n");
        RESET();  /* Perform software reset */
   }
}
```

ROUND

Synopsis

```
#include <math.h>

double round (double x)
```

Description

The **round** function round the argument to the nearest integer value, but in floating-point format. Values midway between integer values are rounded up.

Example

```
#include <math.h>

void
main (void)
{
    double input, rounded;
    input = 1234.5678;
    rounded = round(input);
}
```

See Also

trunc()

SCANF, VSCANF

Synopsis

```
#include <stdio.h>

int scanf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vscanf (const char *, va_list ap)
```

Description

The **scanf()** function performs formatted input ("de-editing") from the *stdin stream*. Similar functions are available for streams in general, and for strings. The function **vscanf()** is similar, but takes a pointer to an argument list rather than a series of additional arguments. This pointer should have been initialised with va_start().

The input conversions are performed according to the **fmt** string; in general a character in the format string must match a character in the input; however a space character in the format string will match zero or more "white space" characters in the input, i.e. *spaces, tabs or newlines*.

A conversion specification takes the form of the character **%**, optionally followed by an assignment suppression character ('*****'), optionally followed by a numerical maximum field width, followed by a conversion specification character. Each conversion specification, unless it incorporates the assignment suppression character, will assign a value to the variable pointed at by the next argument. Thus if there are two conversion specifications in the **fmt** string, there should be two additional pointer arguments.

The conversion characters are as follows:

o x d

Skip white space, then convert a number in base 8, 16 or 10 radix respectively. If a field width was supplied, take at most that many characters from the input. A leading minus sign will be recognized.

s

Skip white space, then copy a maximal length sequence of non-white-space characters. The pointer argument must be a pointer to char. The field width will limit the number of characters copied. The resultant string will be null terminated.

c

Copy the next character from the input. The pointer argument is assumed to be a pointer to char. If a

field width is specified, then copy that many characters. This differs from the **s** format in that white space does not terminate the character sequence.

The conversion characters **o**, **x**, **u**, and **d** may be preceded by an **l** to indicate that the corresponding pointer argument is a pointer to long as appropriate. A preceding **h** will indicate that the pointer argument is a pointer to short rather than int.

Example

```
scanf("%d %s", &a, &c)
     with input " 12s"
     will assign 12 to a, and "s" to s.
```

See Also

fscanf(), sscanf(), printf(), va_arg()

Return Value

The **scanf()** function returns the number of successful conversions; EOF is returned if end-of-file was seen before any conversions were performed.

SETJMP

Synopsis

```
#include <setjmp.h>

int setjmp (jmp_buf buf)
```

Description

The **setjmp()** function is used with longjmp() for non-local goto's. See longjmp() for further information.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf("setjmp returned %d\n", i);
        exit(0);
    }
    printf("setjmp returned 0 - good\n");
    printf("calling inner...\n");
```

```
        inner();
        printf("inner returned - bad!\n");
    }
```

See Also

longjmp()

Return Value

The **setjmp()** function returns zero after the real call, and non-zero if it apparently returns after a call to longjmp().

SIN

Synopsis

```
#include <math.h>

double sin (double f)
```

Description

This function returns the sine function of its argument.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("sin(%3.0f) = %f\n", i, sin(i*C));
        printf("cos(%3.0f) = %f\n", i, cos(i*C));
}
```

See Also

cos(), tan(), asin(), acos(), atan(), atan2()

Return Value

Sine vale of **f**.

SLEEP

Synopsis

```
#include <htc.h>

SLEEP();
```

Description

This macro is used to put the device into a low-power standby mode.

Example

```
#include <htc.h>
extern void init(void);

void
main (void)
{
    init(); /* enable peripherals/interrupts */

    while(1)
        SLEEP();  /* save power while nothing happening */
}
```

SQRT

Synopsis

```
#include <math.h>

double sqrt (double f)
```

Description

The function **sqrt()**, implements a square root routine using Newton's approximation.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i;

    for(i = 0 ; i <= 20.0 ; i += 1.0)
        printf("square root of %.1f = %f\n", i, sqrt(i));
}
```

See Also

exp()

Return Value

Returns the value of the square root.

Note

A domain error occurs if the argument is negative.

SRAND

Synopsis

```
#include <stdlib.h>

void srand (unsigned int seed)
```

Description

The **srand()** function initializes the random number generator accessed by rand() with the given **seed**. This provides a mechanism for varying the starting point of the pseudo-random sequence yielded by rand(). On the Z80, a good place to get a truly random seed is from the refresh register. Otherwise timing a response from the console will do, or just using the system time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

See Also

rand()

STRCAT

Synopsis

```
#include <string.h>

char * strcat (char * s1, const char * s2)
```

Description

This function appends (concatenates) string **s2** to the end of string **s1**. The result will be null terminated. The argument **s1** must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcmp(), strncat(), strlen()

Return Value

The value of **s1** is returned.

STRCHR, STRICHR

Synopsis

```
#include <string.h>

char * strchr (const char * s, int c)
char * strichr (const char * s, int c)
```

Description

The **strchr()** function searches the string **s** for an occurrence of the character **c**. If one is found, a pointer to that character is returned, otherwise NULL is returned.

The **strichr()** function is the case-insensitive version of this function.

Example

```
#include <strings.h>
#include <stdio.h>

void
main (void)
{
    static char temp[] = "Here it is...";
    char c = 's';

    if(strchr(temp, c))
        printf("Character %c was found in string\n", c);
    else
        printf("No character was found in string");
}
```

See Also

strrchr(), strlen(), strcmp()

Return Value

A pointer to the first match found, or NULL if the character does not exist in the string.

Note

Although the function takes an integer argument for the character, only the lower 8 bits of the value are used.

STRCMP, STRICMP

Synopsis

```
#include <string.h>

int strcmp (const char * s1, const char * s2)
int stricmp (const char * s1, const char * s2)
```

Description

The **strcmp()** function compares its two, null terminated, string arguments and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **stricmp()** function is the case-insensitive version of this function.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    int i;

    if((i = strcmp("ABC", "ABc")) < 0)
        printf("ABC is less than ABc\n");
    else if(i > 0)
        printf("ABC is greater than ABc\n");
    else
        printf("ABC is equal to ABc\n");
}
```

See Also

strlen(), strncmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRCPY

Synopsis

```
#include <string.h>

char * strcpy (char * s1, const char * s2)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strncpy(), strlen(), strcat(), strlen()

Return Value

The destination buffer pointer **s1** is returned.

STRCSPN

Synopsis

```
#include <string.h>

size_t strcspn (const char * s1, const char * s2)
```

Description

The **strcspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists of characters NOT from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    static char set[] = "xyz";

    printf("%d\n", strcspn( "abcdevwxyz", set));
    printf("%d\n", strcspn( "xxxbcadefs", set));
    printf("%d\n", strcspn( "1234567890", set));
}
```

See Also

strspn()

Return Value

Returns the length of the segment.

STRLEN

Synopsis

```
#include <string.h>

size_t strlen (const char * s)
```

Description

The **strlen()** function returns the number of characters in the string **s**, not including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

Return Value

The number of characters preceding the null terminator.

STRNCAT

Synopsis

```
#include <string.h>

char * strncat (char * s1, const char * s2, size_t n)
```

Description

This function appends (concatenates) string **s2** to the end of string **s1**. At most **n** characters will be copied, and the result will be null terminated. **s1** must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strncat(s1, s2, 5);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcmp(), strcat(), strlen()

Return Value

The value of **s1** is returned.

STRNCMP, STRNICMP

Synopsis

```
#include <string.h>

int strncmp (const char * s1, const char * s2, size_t n)
int strnicmp (const char * s1, const char * s2, size_t n)
```

Description

The **strncmp()** function compares its two, null terminated, string arguments, up to a maximum of **n** characters, and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **strnicmp()** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int i;

    i = strncmp("abcxyz", "abcxyz",6);
    if(i == 0)
        printf("Both strings are equal\n");
    else if(i > 0)
        printf("String 2 less than string 1\n");
    else
        printf("String 2 is greater than string 1\n");
}
```

See Also

strlen(), strcmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRNCPY

Synopsis

```
#include <string.h>

char * strncpy (char * s1, const char * s2, size_t n)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. At most **n** characters are copied. If string **s2** is longer than **n** then the destination string will not be null terminated. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strncpy(buffer, "Start of line", 6);
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcat(), strlen(), strcmp()

Return Value

The destination buffer pointer **s1** is returned.

STRPBRK

Synopsis

```
#include <string.h>

char * strpbrk (const char * s1, const char * s2)
```

Description

The **strpbrk()** function returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a null pointer if no character from **s2** exists in **s1**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf( "%s\n", str );
        str = strpbrk( str+1, "aeiou" );
    }
}
```

Return Value

Pointer to the first matching character, or NULL if no character found.

STRRCHR, STRRICHR

Synopsis

```
#include <string.h>

char * strrchr (char * s, int c)
char * strrichr (char * s, int c)
```

Description

The **strrchr()** function is similar to the **strchr()** function, but searches from the end of the string rather than the beginning, i.e. it locates the *last* occurrence of the character **c** in the null terminated string **s**. If successful it returns a pointer to that occurrence, otherwise it returns NULL.

The **strrichr()** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf( "%s\n", str );
        str = strrchr( str+1, 's');
    }
}
```

See Also

strchr(), strlen(), strcmp(), strcpy(), strcat()

Return Value

A pointer to the character, or NULL if none is found.

STRSPN

Synopsis

```
#include <string.h>

size_t strspn (const char * s1, const char * s2)
```

Description

The **strspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strspn("This is a string", "This"));
    printf("%d\n", strspn("This is a string", "this"));
}
```

See Also

strcspn()

Return Value

The length of the segment.

STRSTR, STRISTR

Synopsis

```
#include <string.h>

char * strstr (const char * s1, const char * s2)
char * stristr (const char * s1, const char * s2)
```

Description

The **strstr()** function locates the first occurrence of the sequence of characters in the string pointed to by **s2** in the string pointed to by **s1**.

The **stristr()** routine is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strstr("This is a string", "str"));
}
```

Return Value

Pointer to the located string or a null pointer if the string was not found.

STRTOD

Synopsis

```
#include <stdlib.h>

double strtok (const char * s, const char ** res)
```

Description

Parse the string `s` converting it to a double floating point type. This function converts the first occurence of a substring of the input that is made up of characters of the expected form after skipping leading white-space characters. If `res` is not `NULL`, it will be made to point to the first character after the converted sub-string.

Example

```
#include <stdio.h>
#include <strlib.h>

void
main (void)
{
    char buf[] = " 35.7  23.27 ";
    char * end;
    double in1, in2;

    in1 = strtod(buf, &end);
    in2 = strtod(end, NULL);
    printf("in comps: %f, %f\n", in1, in2);
}
```

See Also

atof()

Return Value

Returns a double representing the floating-point value of the converted input string.

STRTOL

Synopsis

```
#include <stdlib.h>

double strtol (const char * s, const char ** res, int base)
```

Description

Parse the string `s` converting it to a long integer type. This function converts the first occurence of a substring of the input that is made up of characters of the expected form after skipping leading white-space characters. The radix of the input is determined from **base**. If this is zero, then the radix defaults to base 10. If **res** is not `NULL`, it will be made to point to the first character after the converted sub-string.

Example

```
#include <stdio.h>
#include <strlib.h>

void
main (void)
{
    char buf[] = " 0X299 0x792 ";
    char * end;
    long in1, in2;

    in1 = strtol(buf, &end, 16);
    in2 = strtol(end, NULL, 16);
    printf("in (decimal): %ld, %ld\n", in1, in2);
}
```

See Also

strtod()

Return Value

Returns a long int representing the value of the converted input string using the specified base.

STRTOK

Synopsis

```
#include <string.h>

char * strtok (char * s1, const char * s2)
```

Description

A number of calls to **strtok()** breaks the string **s1** (which consists of a sequence of zero or more text tokens separated by one or more characters from the separator string **s2**) into its separate tokens.

The first call must have the string **s1**. This call returns a pointer to the first character of the first token, or NULL if no tokens were found. The inter-token separator character is overwritten by a null character, which terminates the current token.

For subsequent calls to **strtok()**, **s1** should be set to a null pointer. These calls start searching from the end of the last token found, and again return a pointer to the first character of the next token, or NULL if no further tokens were found.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;
    char buf[] = "This is a string of words.";
    char * sep_tok = ".,?! ";

    ptr = strtok(buf, sep_tok);
    while(ptr != NULL) {
        printf("%s\n", ptr);
        ptr = strtok(NULL, sep_tok);
    }
}
```

Return Value

Returns a pointer to the first character of a token, or a null pointer if no token was found.

Note

The separator string **s2** may be different from call to call.

TAN

Synopsis

```
#include <math.h>

double tan (double f)
```

Description

The **tan()** function calculates the tangent of **f**.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("tan(%3.0f) = %f\n", i, tan(i*C));
}
```

See Also

sin(), cos(), asin(), acos(), atan(), atan2()

Return Value

The tangent of **f**.

TIME

Synopsis

```
#include <time.h>

time_t time (time_t * t)
```

Description

This function is not provided as it is dependant on the target system supplying the current time. This function will be user implemented. When implemented, this function should return the current time in seconds since 00:00:00 on Jan 1, 1970. If the argument **t** is not equal to NULL, the same value is stored into the object pointed to by **t**.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

ctime(), gmtime(), localtime(), asctime()

Return Value

This routine when implemented will return the current time in seconds since 00:00:00 on Jan 1, 1970.

Note

The **time()** routine is not supplied, if required the user will have to implement this routine to the specifications outlined above.

TOLOWER, TOUPPER, TOASCII

Synopsis

```
#include <ctype.h>

char toupper (int c)
char tolower (int c)
char toascii (int c)
```

Description

The **toupper()** function converts its lower case alphabetic argument to upper case, the **tolower()** routine performs the reverse conversion and the **toascii()** macro returns a result that is guaranteed in the range 0-0177. The functions **toupper()** and **tolower()** return their arguments if it is not an alphabetic character.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void
main (void)
{
    char * array1 = "aBcDE";
    int i;

    for(i=0;i < strlen(array1); ++i) {
        printf("%c", tolower(array1[i]));
    }
    printf("\n");
}
```

See Also

islower(), isupper(), isascii(), et. al.

TRUNC

Synopsis

```
#include <math.h>

double trunc (double x)
```

Description

The **trunc** function rounds the argument to the nearest integer value, in floating-point format, that is not larger in magniture than the argument.

Example

```
#include <math.h>

void
main (void)
{
    double input, rounded;
    input = 1234.5678;
    rounded = trunc(input);
}
```

See Also

round()

UDIV

Synopsis

```
#include <stdlib.h>

int udiv (unsigned num, unsigned demon)
```

Description

The **udiv()** function calculate the quotient and remainder of the division of `number` and `denom`, storing the results into a udiv_t structure which is returned.

Example

```
#include <stdlib.h>

void
main (void)
{
    udiv_t result;
    unsigned  num = 1234, den = 7;

    result = udiv(num, den);
}
```

See Also

uldiv(), div(), ldiv()

Return Value

Returns the the quotient and remainder as a udiv_t structure.

ULDIV

Synopsis

```
#include <stdlib.h>

int uldiv (unsigned long num, unsigned long demon)
```

Description

The **uldiv()** function calculate the quotient and remainder of the division of `number` and `denom`, storing the results into a uldiv_t structure which is returned.

Example

```
#include <stdlib.h>

void
main (void)
{
    uldiv_t result;
    unsigned long num = 1234, den = 7;

    result = uldiv(num, den);
}
```

See Also

ldiv(), udiv(), div()

Return Value

Returns the the quotient and remainder as a uldiv_t structure.

UNGETCH

Synopsis

```
#include <conio.h>

void ungetch (char c)
```

Description

The **ungetch()** function will push back the character **c** onto the console stream, such that a subsequent getch() operation will return the character. At most one level of push back will be allowed.

See Also

getch(), getche()

UTOA

Synopsis

```
#include <stdlib.h>

char * utoa (char * buf, unsigned val, int base)
```

Description

The function **utoa** converts the unsigned contents of **val** into a string which is stored into **buf**. The conversion is performed according to the radix specified in **base**. **buf** is assumed to reference a buffer which has sufficent space allocated to it.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    utoa(buf, 1234, 16);
    printf("The buffer holds %s\n", buf);
}
```

See Also

strtol(), itoa(), ltoa(), ultoa()

Return Value

This routine returns a copy of the buffer into which the result is written.

VA_START, VA_ARG, VA_END

Synopsis

```
#include <stdarg.h>

void va_start (va_list ap, parmN)
type va_arg  (ap, type)
void va_end  (va_list ap)
```

Description

These macros are provided to give access in a portable way to parameters to a function represented in a prototype by the ellipsis symbol (...), where type and number of arguments supplied to the function are not known at compile time.

The rightmost parameter to the function (shown as **parmN**) plays an important role in these macros, as it is the starting point for access to further parameters. In a function taking variable numbers of arguments, a variable of type **va_list** should be declared, then the macro **va_start()** invoked with that variable and the name of **parmN**. This will initialize the variable to allow subsequent calls of the macro **va_arg()** to access successive parameters.

Each call to **va_arg()** requires two arguments; the variable previously defined and a type name which is the type that the next parameter is expected to be. Note that any arguments thus accessed will have been widened by the default conventions to *int, unsigned int* or *double*. For example if a character argument has been passed, it should be accessed by **va_arg(ap, int)** since the *char* will have been widened to *int*.

An example is given below of a function taking one integer parameter, followed by a number of other parameters. In this example the function expects the subsequent parameters to be pointers to char, but note that the compiler is not aware of this, and it is the programmers responsibility to ensure that correct arguments are supplied.

Example

```
#include <stdio.h>
#include <stdarg.h>

void
pf (int a, ...)
{
```

```
    va_list ap;

    va_start(ap, a);
    while(a--)
        puts(va_arg(ap, char *));
    va_end(ap);
}

void
main (void)
{
    pf(3, "Line 1", "line 2", "line 3");
}
```

WRITETIMER*x*

Synopsis

```
#include <htc.h>

WRITETIMER0(unsigned int);
WRITETIMER1(unsigned int);
WRITETIMER3(unsigned int);
```

Description

The **WRITETIMER0**, **WRITETIMER1()** and **WRITETIMER3()** macros will assign a 16-Bit value to the TMRxL and TMRxH register pair of the corresponding device timer. Using this macro will ensure that the bytes are written in the correct order.

Example

```
#include <htc.h>

void
main (void)
{
    WRITETIMER1(0xF500);
}
```

See Also

READTIMER*x*()

XTOI

Synopsis

```
#include <stdlib.h>

unsigned xtoi (const char * s)
```

Description

The **xtoi()** function scans the character string passed to it, skipping leading blanks reading an optional sign, and converts an ASCII representation of a hexadecimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = xtoi(buf);
    printf("Read %s: converted to %x\n", buf, i);
}
```

See Also

atoi()

Return Value

An unsigned integer. If no number is found in the string, zero will be returned.

Appendix B

Error and Warning Messages

This chapter lists most error, warning and advisory messages from all HI-TECH C compilers, with an explanation of each message. Most messages have been assigned a unique number which appears in brackets before each message in this chapter, and which is also printed by the compiler when the message is issued. The messages shown here are sorted by their number. Un-numbered messages appear toward the end and are sorted alphabetically.

The name of the application(s) that could have produced the messages are listed in brackets opposite the error message. In some cases examples of code or options that could trigger the error are given. The use of * in the error message is used to represent a string that the compiler will substitute that is specific to that particular error.

Note that one problem in your C or assembler source code may trigger more than one error message. You should attempt to resolve errors or warnings in the order in which they are produced.

(1) too many errors (*) ***(all applications)***

The executing compiler application has encountered too many errors and will exit immediately. Other uncompiled source files will be processed, but the compiler applications that would normally be executed in due course will not be run. The number of errors that can be accepted can be controlled using the `--ERRORS` option, See Section 2.6.32.

(2) error/warning (*) generated, but no description available ***(all applications)***

The executing compiler application has emitted a message (advisory/warning/error), but there is no description available in the message description file (MDF) to print. This may be because the MDF is out of date, or the message issue has not been tranlated into the selected language.

(3) malformed error information on line *, in file * ***(all applications)***

The compiler has attempted to load the messages for the selected language, but the message description file (MDF) was corrupted and could not be read correctly.

(100) unterminated #if[n][def] block from line * ***(Preprocessor)***

A `#if` or similar block was not terminated with a matching `#endif`, e.g.:

```
#if INPUT        /* error flagged here */
void main(void)
{
  run();
}                /* no #endif was found in this module */
```

(101) #* may not follow #else ***(Preprocessor)***

A `#else` or `#elif` has been used in the same conditional block as a `#else`. These can only follow a `#if`, e.g.:

```
#ifdef FOO
  result = foo;
#else
  result = bar;
#elif defined(NEXT)   /* the #else above terminated the #if */
  result = next(0);
#endif
```

(102) #* must be in an #if ***(Preprocessor)***

The `#elif`, `#else` or `#endif` directive must be preceded by a matching `#if` line. If there is an apparently corresponding `#if` line, check for things like extra `#endif`'s, or improperly terminated comments, e.g.:

```
#ifdef FOO
  result = foo;
#endif
  result = bar;
#elif defined(NEXT)   /* the #endif above terminated the #if */
  result = next(0);
#endif
```

(103) #error: * ***(Preprocessor)***

This is a programmer generated error; there is a directive causing a deliberate error. This is normally used to check compile time defines etc. Remove the directive to remove the error, but first check as to why the directive is there.

(104) preprocessor #assert failure ***(Preprocessor)***

The argument to a preprocessor `#assert` directive has evaluated to zero. This is a programmer induced error.

```
#assert SIZE == 4  /* size should never be 4 */
```

(105) no #asm before #endasm ***(Preprocessor)***

A `#endasm` operator has been encountered, but there was no previous matching `#asm`, e.g.:

```
void cleardog(void)
{
  clrwdt
#endasm  /* in-line assembler ends here,
            only where did it begin? */
}
```

(106) nested #asm directives ***(Preprocessor)***

It is not legal to nest `#asm` directives. Check for a missing or misspelt `#endasm` directive, e.g.:

```
#asm
  move r0, #0aah
#asm      ; previous #asm must be closed before opening another
  sleep
#endasm
```

(107) illegal # directive "*" ***(Preprocessor, Parser)***

The compiler does not understand the `#` directive. It is probably a misspelling of a pre-processor `#` directive, e.g.:

```
#indef DEBUG  /* oops -- that should be #undef DEBUG */
```

(108) #if[n][def] without an argument *(Preprocessor)*

The preprocessor directives `#if`, `#ifdef` and `#ifndef` must have an argument. The argument to `#if` should be an expression, while the argument to `#ifdef` or `#ifndef` should be a single name, e.g.:

```
#if            /* oops -- no argument to check */
  output = 10;
#else
  output = 20;
#endif
```

(109) #include syntax error *(Preprocessor)*

The syntax of the filename argument to `#include` is invalid. The argument to `#include` must be a valid file name, either enclosed in double quotes `""` or angle brackets `< >`. Spaces should not be included, and the closing quote or bracket must be present. There should be nothing else on the line other than comments, e.g.:

```
#include stdio.h  /* oops -- should be: #include <stdio.h> */
```

(110) too many file arguments; usage: cpp [input [output]] *(Preprocessor)*

`CPP` should be invoked with at most two file arguments. Contact HI-TECH Support if the preprocessor is being executed by a compiler driver.

(111) redefining preprocessor macro "*" *(Preprocessor)*

The macro specified is being redefined, to something different to the original definition. If you want to deliberately redefine a macro, use #`undef` first to remove the original definition, e.g.:

```
#define ONE 1
/* elsewhere: */
/* Is this correct? It will overwrite the first definition. */
#define ONE one
```

(112) #define syntax error *(Preprocessor)*

A macro definition has a syntax error. This could be due to a macro or formal parameter name that does not start with a letter or a missing *closing parenthesis* , `)`, e.g.:

```
#define FOO(a, 2b)  bar(a, 2b)  /* 2b is not to be! */
```

(113) unterminated string in preprocessor macro body ***(Preprocessor, Assembler)***

A macro definition contains a string that lacks a closing quote.

(114) illegal #undef argument ***(Preprocessor)***

The argument to `#undef` must be a valid name. It must start with a letter, e.g.:

```
#undef 6YYY  /* this isn't a valid symbol name */
```

(115) recursive preprocessor macro definition of "*" defined by "*" ***(Preprocessor)***

The named macro has been defined in such a manner that expanding it causes a recursive expansion of itself!

(116) end of file within preprocessor macro argument from line * ***(Preprocessor)***

A macro argument has not been terminated. This probably means the closing parenthesis has been omitted from a macro invocation. The line number given is the line where the macro argument started, e.g.:

```
#define FUNC(a, b) func(a+b)
FUNC(5, 6;    /* oops -- where is the closing bracket? */
```

(117) misplaced constant in #if ***(Preprocessor)***

A constant in a `#if` expression should only occur in syntactically correct places. This error is most probably caused by omission of an operator, e.g.:

```
#if FOO BAR  /* oops -- did you mean: #if FOO == BAR ? */
```

(118) stack overflow processing #if expression ***(Preprocessor)***

The preprocessor filled up its expression evaluation stack in a `#if` expression. Simplify the expression — it probably contains too many parenthesized subexpressions.

(119) invalid expression in #if line ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(120) operator "*" in incorrect context ***(Preprocessor)***

An operator has been encountered in a `#if` expression that is incorrectly placed, e.g. two binary operators are not separated by a value, e.g.:

```
#if FOO * % BAR == 4  /* what is "* %" ? */
  #define BIG
#endif
```

(121) expression stack overflow at operator "*" ***(Preprocessor)***

Expressions in `#if` lines are evaluated using a stack with a size of 128. It is possible for very complex expressions to overflow this. Simplify the expression.

(122) unbalanced parenthesis at operator "*" ***(Preprocessor)***

The evaluation of a `#if` expression found mismatched parentheses. Check the expression for correct parenthesisation, e.g.:

```
#if ((A) + (B)  /* oops -- a missing ), I think */
  #define ADDED
#endif
```

(123) misplaced "?" or ":"; previous operator is "*" ***(Preprocessor)***

A colon operator has been encountered in a `#if` expression that does not match up with a corresponding `?` operator, e.g.:

```
#if XXX : YYY  /* did you mean:  #if COND ? XXX : YYY */
```

(124) illegal character "*" in #if ***(Preprocessor)***

There is a character in a `#if` expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
#if `YYY`  /* what are these characters doing here? */
  int m;
#endif
```

(125) illegal character (* decimal) in #if *(Preprocessor)*

There is a non-printable character in a `#if` expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
#if ^SYYY  /* what is this control characters doing here? */
  int m;
#endif
```

(126) strings can't be used in #if *(Preprocessor)*

The preprocessor does not allow the use of strings in `#if` expressions, e.g.:

```
/* no string operations allowed by the preprocessor */
#if MESSAGE > "hello"
#define DEBUG
#endif
```

(127) bad syntax for defined() in #[el]if *(Preprocessor)*

The `defined()` pseudo-function in a preprocessor expression requires its argument to be a single name. The name must start with a letter and should be enclosed in parentheses, e.g.:

```
/* oops -- defined expects a name, not an expression */
#if defined(a&b)
  input = read();
#endif
```

(128) illegal operator in #if *(Preprocessor)*

A `#if` expression has an illegal operator. Check for correct syntax, e.g.:

```
#if FOO = 6  /* oops -- should that be: #if FOO == 5 ? */
```

(129) unexpected "\" in #if *(Preprocessor)*

The *backslash* is incorrect in the `#if` statement, e.g.:

```
#if FOO == \34
  #define BIG
#endif
```

(130) unknown type "*" in #[el]if sizeof() ***(Preprocessor)***

An unknown type was used in a preprocessor `sizeof()`. The preprocessor can only evaluate `sizeof()` with basic types, or pointers to basic types, e.g.:

```
#if sizeof(unt) == 2  /* should be: #if sizeof(int) == 2 */
  i = 0xFFFF;
#endif
```

(131) illegal type combination in #[el]if sizeof() ***(Preprocessor)***

The preprocessor found an illegal type combination in the argument to `sizeof()` in a `#if` expression, e.g.

```
/* To sign, or not to sign, that is the error. */
#if sizeof(signed unsigned int) == 2
  i = 0xFFFF;
#endif
```

(132) no type specified in #[el]if sizeof() ***(Preprocessor)***

Sizeof() was used in a preprocessor `#if` expression, but no type was specified. The argument to `sizeof()` in a preprocessor expression must be a valid simple type, or pointer to a simple type, e.g.:

```
#if sizeof()  /* oops -- size of what? */
  i = 0;
#endif
```

(133) unknown type code (0x*) in #[el]if sizeof() ***(Preprocessor)***

The preprocessor has made an internal error in evaluating a `sizeof()` expression. Check for a malformed type specifier. This is an internal error. Contact HI-TECH Software technical support with details.

(134) syntax error in #[el]if sizeof() ***(Preprocessor)***

The preprocessor found a syntax error in the argument to `sizeof`, in a `#if` expression. Probable causes are mismatched parentheses and similar things, e.g.:

```
#if sizeof(int == 2)  // oops - should be: #if sizeof(int) == 2
  i = 0xFFFF;
#endif
```

(135) unknown operator (*) in #if ***(Preprocessor)***

The preprocessor has tried to evaluate an expression with an operator it does not understand. This is an internal error. Contact HI-TECH Software technical support with details.

(137) strange character "*" after ## ***(Preprocessor)***

A character has been seen after the token catenation operator `##` that is neither a letter nor a digit. Since the result of this operator must be a legal token, the operands must be tokens containing only letters and digits, e.g.:

```
/* the ’ character will not lead to a valid token */
#define cc(a, b) a ## ’b
```

(138) strange character (*) after ## ***(Preprocessor)***

An unprintable character has been seen after the token catenation operator `##` that is neither a letter nor a digit. Since the result of this operator must be a legal token, the operands must be tokens containing only letters and digits, e.g.:

```
/* the ’ character will not lead to a valid token */
#define cc(a, b) a ## ’b
```

(139) end of file in comment ***(Preprocessor)***

End of file was encountered inside a comment. Check for a missing closing comment flag, e.g.:

```
   /* Here the comment begins. I’m not sure where I end, though
}
```

(140) can’t open * file "*": * ***(Driver, Preprocessor, Code Generator, Assembler)***

The command file specified could not be opened for reading. Confirm the spelling and path of the file specified on the command line, e.g.:

```
picc @communds
```

should that be:

```
picc @commands
```

(141) can't open * file "*": * *(Any)*

An output file could not be created. Confirm the spelling and path of the file specified on the command line.

(144) too many nested #if blocks *(Preprocessor)*

`#if`, `#ifdef` etc. blocks may only be nested to a maximum of 32.

(146) #include filename too long *(Preprocessor)*

A filename constructed while looking for an include file has exceeded the length of an internal buffer. Since this buffer is 4096 bytes long, this is unlikely to happen.

(147) too many #include directories specified *(Preprocessor)*

A maximum of 7 directories may be specified for the preprocessor to search for include files. The number of directories specified with the driver is too great.

(148) too many arguments for preprocessor macro *(Preprocessor)*

A macro may only have up to 31 parameters, as per the C Standard.

(149) preprocessor macro work area overflow *(Preprocessor)*

The total length of a macro expansion has exceeded the size of an internal table. This table is normally 32768 bytes long. Thus any macro expansion must not expand into a total of more than 32K bytes.

(150) illegal "__" preprocessor macro "*" *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(151) too many arguments in preprocessor macro expansion *(Preprocessor)*

There were too many arguments supplied in a macro invocation. The maximum number allowed is 31.

(152) bad dp/nargs in openpar(): c = * *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(153) out of space in preprocessor macro "*" argument expansion ***(Preprocessor)***

A macro argument has exceeded the length of an internal buffer. This buffer is normally 4096 bytes long.

(155) work buffer overflow conatenating "*" ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(156) work buffer "*" overflow ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(157) can't allocate * bytes of memory ***(Code Generator, Assembler, Optimiser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(158) invalid disable in preprocessor macro "*" ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(159) too many calls to unget() ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(161) control line "*" within preprocessor macro expansion ***(Preprocessor)***

A preprocessor control line (one starting with a #) has been encountered while expanding a macro. This should not happen.

(162) #warning: * ***(Preprocessor, Driver)***

This warning is either the result of user-defined `#warning` preprocessor directive or the driver encountered a problem reading the the map file. If the latter then please HI-TECH Software technical support with details

(163) unexpected text in control line ignored ***(Preprocessor)***

This warning occurs when extra characters appear on the end of a control line, e.g. The extra text will be ignored, but a warning is issued. It is preferable (and in accordance with Standard C) to enclose the text as a comment, e.g.:

```
#if defined(END)
  #define NEXT
#endif END     /* END would be better in a comment here */
```

(164) #include filename "*" was converted to lower case ***(Preprocessor)***

The `#include` file name had to be converted to lowercase before it could be opened, e.g.:

```
#include <STDIO.H>  /* oops -- should be: #include <stdio.h> */
```

(165) #include filename "*" does not match actual name (check upper/lower case) ***(Preprocessor)***

In Windows versions this means the file to be included actually exists and is spelt the same way as the `#include` filename, however the case of each does not exactly match. For example, specifying `#include "code.c"` will include `Code.c` if it is found. In Linux versions this warning could occur if the file wasn't found.

(166) too few values specified with option "*" ***(Preprocessor)***

The list of values to the preprocessor (CPP) `-S` option is incomplete. This should not happen if the preprocessor is being invoked by the compiler driver. The values passes to this option represent the sizes of `char`, `short`, `int`, `long`, `float` and `double` types.

(167) too many values specified with -S option; "*" unused ***(Preprocessor)***

There were too many values supplied to the -S preprocessor option. See the Error Message `-s, too few values specified in *` on page 370.

(168) unknown option "*" ***(Any)***

This option given to the component which caused the error is not recognized.

(169) strange character (*) after ## ***(Preprocessor)***

There is an unexpected character after `#`.

(170) symbol "*" in undef was never defined ***(Preprocessor)***

The symbol supplied as argument to `#undef` was not already defined. This warning may be disabled with some compilers. This warning can be avoided with code like:

```
#ifdef SYM
  #undef SYM  /* only undefine if defined */
#endif
```

(171) wrong number of preprocessor macro arguments for "*" (* instead of *)* ***(Preprocessor)***

A macro has been invoked with the wrong number of arguments, e.g.:

```
#define ADD(a, b) (a+b)
ADD(1, 2, 3)        /* oops -- only two arguments required */
```

(172) formal parameter expected after # ***(Preprocessor)***

The stringization operator `#` (not to be confused with the leading `#` used for preprocessor control lines) must be followed by a formal macro parameter, e.g.:

```
#define str(x) #y  /* oops -- did you mean x instead of y? */
```

If you need to stringize a token, you will need to define a special macro to do it, e.g.

```
#define __mkstr__(x) #x
```

then use `__mkstr__(token)` wherever you need to convert a token into a string.

(173) undefined symbol "*" in #if, 0 used ***(Preprocessor)***

A symbol on a `#if` expression was not a defined preprocessor macro. For the purposes of this expression, its value has been taken as zero. This warning may be disabled with some compilers. Example:

```
#if FOO+BAR    /* e.g. FOO was never #defined */
  #define GOOD
#endif
```

(174) multi-byte constant "*" isn't portable ***(Preprocessor)***

Multi-byte constants are not portable, and in fact will be rejected by later passes of the compiler, e.g.:

```
#if CHAR == 'ab'
  #define MULTI
#endif
```

(175) division by zero in #if; zero result assumed ***(Preprocessor)***

Inside a `#if` expression, there is a division by zero which has been treated as yielding zero, e.g.:

```
#if foo/0  /* divide by 0: was this what you were intending? */
  int a;
#endif
```

(176) missing newline ***(Preprocessor)***

A new line is missing at the end of the line. Each line, including the last line, must have a new line at the end. This problem is normally introduced by editors.

(177) symbol "*" in -U option was never defined ***(Preprocessor)***

A macro name specified in a `-U` option to the preprocessor was not initially defined, and thus cannot be undefined.

(179) nested comments ***(Preprocessor)***

This warning is issued when nested comments are found. A nested comment may indicate that a previous closing comment marker is missing or malformed, e.g.:

```
output = 0;  /* a comment that was left unterminated
flag = TRUE; /* next comment:
                hey, where did this line go? */
```

(180) unterminated comment in included file ***(Preprocessor)***

Comments begun inside an included file must end inside the included file.

(181) non-scalar types can't be converted to other types *(Parser)*

You can't convert a structure, union or array to another type, e.g.:

```
struct TEST test;
struct TEST * sp;
sp = test;       /* oops -- did you mean: sp = &test; ? */
```

(182) illegal conversion between types *(Parser)*

This expression implies a conversion between incompatible types, e.g. a conversion of a structure type into an integer, e.g.:

```
struct LAYOUT layout;
int i;
layout = i;          /* int cannot be converted to struct */
```

Note that even if a structure only contains an `int`, for example, it cannot be assigned to an `int` variable, and vice versa.

(183) function or function pointer required *(Parser)*

Only a function or function pointer can be the subject of a function call, e.g.:

```
int a, b, c, d;
a = b(c+d);   /* b is not a function --
                 did you mean a = b*(c+d) ? */
```

(184) calling an interrupt function is illegal *(Parser)*

A function qualified `interrupt` can't be called from other functions. It can only be called by a hardware (or software) interrupt. This is because an `interrupt` function has special function entry and exit code that is appropriate only for calling from an interrupt. An `interrupt` function can call other non-`interrupt` functions.

(185) function does not take arguments *(Parser, Code Generator)*

This function has no parameters, but it is called here with one or more arguments, e.g.:

```
int get_value(void);
void main(void)
{
  int input;
  input = get_value(6);  /* oops --
                            parameter should not be here */
}
```

(186) too many function arguments ***(Parser)***

This function does not accept as many arguments as there are here.

```
void add(int a, int b);
add(5, 7, input);      /* call has too many arguments */
```

(187) too few function arguments ***(Parser)***

This function requires more arguments than are provided in this call, e.g.:

```
void add(int a, int b);
add(5);                 /* this call needs more arguments */
```

(188) constant expression required ***(Parser)***

In this context an expression is required that can be evaluated to a constant at compile time, e.g.:

```
int a;
switch(input) {
  case a:  /* oops!
              can't use variable as part of a case label */
    input++;
}
```

(189) illegal type for array dimension ***(Parser)***

An array dimension must be either an integral type or an enumerated value.

```
int array[12.5];  /* oops -- twelve and a half elements, eh? */
```

(190) illegal type for index expression ***(Parser)***

An index expression must be either integral or an enumerated value, e.g.:

```
int i, array[10];
i = array[3.5];   /* oops --
                     exactly which element do you mean? */
```

(191) cast type must be scalar or void ***(Parser)***

A typecast (an abstract type declarator enclosed in parentheses) must denote a type which is either scalar (i.e. not an array or a structure) or the type `void`, e.g.:

```
lip = (long [])input;  /* oops -- maybe: lip = (long *)input */
```

(192) undefined identifier "*" ***(Parser)***

This symbol has been used in the program, but has not been defined or declared. Check for spelling errors if you think it has been defined.

(193) not a variable identifier "*" ***(Parser)***

This identifier is not a variable; it may be some other kind of object, e.g. a label.

(194) ")" expected ***(Parser)***

A *closing parenthesis*, `)`, was expected here. This may indicate you have left out this character in an expression, or you have some other syntax error. The error is flagged on the line at which the code first starts to make no sense. This may be a statement following the incomplete expression, e.g.:

```
if(a == b  /* the closing parenthesis is missing here */
  b = 0;   /* the error is flagged here */
```

(195) expression syntax ***(Parser)***

This expression is badly formed and cannot be parsed by the compiler, e.g.:

```
a /=% b;  /* oops -- maybe that should be: a /= b; */
```

(196) struct/union required ***(Parser)***

A structure or union identifier is required before a dot `.`, e.g.:

```
int a;
a.b = 9;  /* oops -- a is not a structure */
```

(197) struct/union member expected ***(Parser)***

A structure or union member name must follow a dot (".") or arrow ("->").

(198) undefined struct/union "*" ***(Parser)***

The specified structure or union tag is undefined, e.g.

```
struct WHAT what;  /* a definition for WHAT was never seen */
```

(199) logical type required ***(Parser)***

The expression used as an operand to `if`, `while` statements or to boolean operators like `!` and `&&` must be a scalar integral type, e.g.:

```
struct FORMAT format;
if(format)            /* this operand must be a scaler type */
  format.a = 0;
```

(200) taking the address of a register variable is illegal ***(Parser)***

A variable declared `register` may not have storage allocated for it in memory, and thus it is illegal to attempt to take the address of it by applying the `&` operator, e.g.:

```
int * proc(register int in)
{
  int * ip = &in;
  /* oops -- in may not have an address to take */
  return ip;
}
```

(201) taking the address of this object is illegal ***(Parser)***

The expression which was the operand of the `&` operator is not one that denotes memory storage ("an lvalue") and therefore its address can not be defined, e.g.:

```
ip = &8;  /* oops -- you can't take the address of a literal */
```

(202) only lvalues may be assigned to or modified *(Parser)*

Only an lvalue (i.e. an identifier or expression directly denoting addressable storage) can be assigned to or otherwise modified, e.g.:

```
int array[10];
int * ip;
char c;
array = ip;   /* array isn't a variable,
                 it can't be written to */
```

A typecast does not yield an lvalue, e.g.:

```
/* the contents of c cast to int
   is only a intermediate value */
(int)c = 1;
```

However you can write this using pointers:

```
*(int *)&c = 1
```

(203) illegal operation on bit variable *(Parser)*

Not all operations on `bit` variables are supported. This operation is one of those, e.g.:

```
bit   b;
int * ip;
ip = &b;  /* oops --
             cannot take the address of a bit object */
```

(204) void function can't return a value *(Parser)*

A void function cannot return a value. Any `return` statement should not be followed by an expression, e.g.:

```
void run(void)
{
  step();
  return 1;
  /* either run should not be void, or remove the 1 */
}
```

(205) integral type required ***(Parser)***

This operator requires operands that are of integral type only.

(206) illegal use of void expression ***(Parser)***

A `void` expression has no value and therefore you can't use it anywhere an expression with a value is required, e.g. as an operand to an arithmetic operator.

(207) simple type required for "*" ***(Parser)***

A simple type (i.e. not an array or structure) is required as an operand to this operator.

(208) operands of "*" not same type ***(Parser)***

The operands of this operator are of different pointer, e.g.:

```
int * ip;
char * cp, * cp2;
cp = flag ? ip : cp2;
/* result of ? : will be int * or char * */
```

Maybe you meant something like:

```
cp = flag ? (char *)ip : cp2;
```

(209) type conflict ***(Parser)***

The operands of this operator are of incompatible types.

(210) bad size list ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(211) taking sizeof bit is illegal ***(Parser)***

It is illegal to use the `sizeof` operator with the HI-TECH C `bit` type. When used against a type the `sizeof` operator gives the number of bytes required to store an object that type. Therefore its usage with the `bit` type make no sense and is an illegal operation.

(212) missing number after pragma "pack" ***(Parser)***

The pragma `pack` requires a decimal number as argument. This specifies the alignment of each member within the structure. Use this with caution as some processors enforce alignment and will not operate correctly if word fetches are made on odd boundaries, e.g.:

```
#pragma pack  /* what is the alignment value */
```

Maybe you meant something like:

```
#pragma pack 2
```

(214) missing number after pragma "interrupt_level" ***(Parser)***

The pragma `interrupt_level` requires an argument from 0 to 7.

(215) missing argument to pragma "switch" ***(Parser)***

The pragma switch requires an argument of `auto`, `direct` or `simple`, e.g.:

```
#pragma switch  /* oops -- this requires a switch mode */
```

maybe you meant something like:

```
#pragma switch simple
```

(216) missing argument to pragma "psect" ***(Parser)***

The pragma `psect` requires an argument of the form *`oldname=newname`* where *`oldname`* is an existing psect name known to the compiler, and *`newname`* is the desired new name, e.g.:

```
#pragma psect  /* oops -- this requires an psect to redirect */
```

maybe you meant something like:

```
#pragma psect text=specialtext
```

(218) missing name after pragma "inline" ***(Parser)***

The `inline` pragma expects the name of a function to follow. The function name must be recognized by the code generator for it to be expanded; other functions are not altered, e.g.:

```
#pragma inline  /* what is the function name? */
```

maybe you meant something like:

```
#pragma inline memcpy
```

(219) missing name after pragma "printf_check" ***(Parser)***

The `printf_check` pragma expects the name of a function to follow. This specifies printf-style format string checking for the function, e.g.

```
#pragma printf_check  /* what function is to be checked? */
```

Maybe you meant something like:

```
#pragma printf_check sprintf
```

Pragmas for all the standard printf-like function are already contained in `<stdio.h>`.

(220) exponent expected ***(Parser)***

A floating point constant must have at least one digit after the `e` or `E`., e.g.:

```
float f;
f = 1.234e;  /* oops -- what is the exponent? */
```

(221) hexadecimal digit expected ***(Parser)***

After `0x` should follow at least one of the hex digits `0-9` and `A-F` or `a-f`, e.g.:

```
a = 0xg6;  /* oops -- was that meant to be a = 0xf6 ? */
```

(222) binary digit expected ***(Parser)***

A binary digit was expected following the `0b` format specifier, e.g.

```
i = 0bf000;  /* wooops -- f000 is not a base two value */
```

(223) digit out of range ***(Parser, Assembler, Optimiser)***

A digit in this number is out of range of the radix for the number, e.g. using the digit 8 in an octal number, or hex digits A-F in a decimal number. An octal number is denoted by the digit string commencing with a zero, while a hex number starts with "0X" or "0x". For example:

```
int a = 058;
/* leading 0 implies octal which has digits 0 - 7 */
```

(224) illegal "#" directive ***(Parser)***

An illegal # preprocessor has been detected. Likely a directive has been misspelt in your code somewhere.

(225) missing character in character constant **(Parser)**

The character inside the single quotes is missing, e.g.:

```
char c = ’’;  /* the character value of what? */
```

(226) char const too long ***(Parser)***

A character constant enclosed in single quotes may not contain more than one character, e.g.:

```
c = ’12’;  /* oops -- only one character may be specified */
```

(227) "." expected after ".." ***(Parser)***

The only context in which two successive dots may appear is as part of the *ellipsis* symbol, which must have 3 dots. (An *ellipsis* is used in function prototypes to indicate a variable number of parameters.)

Either `..` was meant to be an *ellipsis* symbol which would require you to add an extra dot, or it was meant to be a *structure member operator* which would require you remove one dot.

(228) illegal character (0x*) ***(Parser)***

This character is illegal in the C code. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
c = `a`;  /* oops -- did you mean c = ’a’; ? */
```

(229) unknown qualifier "*" given to -A ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(230) missing argument to -A ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(231) unknown qualifier "*" given to -I ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(232) missing argument to -I ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(233) bad -Q option "*" ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(234) close error ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(236) simple integer expression required ***(Parser)***

A simple integral expression is required after the operator `@`, used to associate an absolute address with a variable, e.g.:

```
int address;
char LOCK @ address;
```

(237) function "*" redefined ***(Parser)***

More than one definition for a function has been encountered in this module. Function overloading is illegal, e.g.:

```
int twice(int a)
{
  return a*2;
}
/* only one prototype & definition of rv can exist */
long twice(long a)
{
  return a*2;
}
```

(238) illegal initialisation *(Parser)*

You can't initialise a `typedef` declaration, because it does not reserve any storage that can be initialised, e.g.:

```
/* oops -- uint is a type, not a variable */
typedef unsigned int uint = 99;
```

(239) identifier "*" redefined (from line *) *(Parser)*

This identifier has already been defined in the same scope. It cannot be defined again, e.g.:

```
int a;  /* a filescope variable called "a" */
int a;  /* attempting to define another of the same name */
```

Note that variables with the same name, but defined with different scopes are legal, but not recommended.

(240) too many initializers *(Parser)*

There are too many initializers for this object. Check the number of initializers against the object definition (array or structure), e.g.:

```
/* three elements, but four initializers */
int ivals[3] = { 2, 4, 6, 8};
```

(241) initialization syntax *(Parser)*

The initialisation of this object is syntactically incorrect. Check for the correct placement and number of braces and commas, e.g.:

```
int iarray[10] = {{'a', 'b', 'c'};
/* oops -- one two many {s */
```

(242) illegal type for switch expression *(Parser)*

A `switch` operation must have an expression that is either an integral type or an enumerated value, e.g:

```
double d;
switch(d) {  /* oops -- this must be integral */
  case '1.0':
    d = 0;
}
```

(243) inappropriate break/continue *(Parser)*

A `break` or `continue` statement has been found that is not enclosed in an appropriate control structure. A `continue` can only be used inside a `while`, `for` or `do while` loop, while `break` can only be used inside those loops or a `switch` statement, e.g.:

```
switch(input) {
  case 0:
    if(output == 0)
      input = 0xff;
    } /* oops! this shouldn't be here and closed the switch */
    break;     /* this should be inside the switch */
```

(244) "default" case redefined *(Parser)*

There is only allowed to be one `default` label in a switch statement. You have more than one, e.g.:

```
switch(a) {
default:     /* if this is the default case... */
  b = 9;
  break;
default:     /* then what is this? */
  b = 10;
  break;
```

(245) "default" case not in switch *(Parser)*

A label has been encountered called `default` but it is not enclosed by a `switch` statement. A `default` label is only legal inside the body of a `switch` statement.

If there is a `switch` statement before this `default` label, there may be one too many closing braces in the `switch` code which would prematurely terminate the `switch` statement. See example for Error Message `'case' not in switch` on page 384.

(246) case label not in switch *(Parser)*

A `case` label has been encountered, but there is no enclosing `switch` statement. A `case` label may only appear inside the body of a `switch` statement.

If there is a `switch` statement before this `case` label, there may be one too many closing braces in the `switch` code which would prematurely terminate the `switch` statement, e.g.:

```
switch(input) {
  case '0':
    count++;
    break;
  case '1':
    if(count>MAX)
      count= 0;
    }          /* oops -- this shouldn't be here */
    break;
  case '2':    /* error flagged here */
```

(247) duplicate label "*" ***(Parser)***

The same name is used for a label more than once in this function. Note that the scope of labels is the entire function, not just the block that encloses a label, e.g.:

```
start:
  if(a > 256)
    goto end;
start:           /* error flagged here */
  if(a == 0)
    goto start;  /* which start label do I jump to? */
```

(248) inappropriate "else" ***(Parser)***

An `else` keyword has been encountered that cannot be associated with an `if` statement. This may mean there is a missing brace or other syntactic error, e.g.:

```
/* here is a comment which I have forgotten to close...
if(a > b) {
  c = 0;
/* ... that will be closed here, thus removing the "if" */
else        /* my "if" has been lost */
  c = 0xff;
```

(249) probable missing "}" in previous block ***(Parser)***

The compiler has encountered what looks like a function or other declaration, but the preceding function has not been ended with a closing brace. This probably means that a closing brace has been omitted from somewhere in the previous function, although it may well not be the last one, e.g.:

```
void set(char a)
{
  PORTA = a;
                  /* the closing brace was left out here */
void clear(void)  /* error flagged here */
{
  PORTA = 0;
}
```

(251) array dimension redeclared ***(Parser)***

An array dimension has been declared as a different non-zero value from its previous declaration. It is acceptable to redeclare the size of an array that was previously declared with a zero dimension, but not otherwise, e.g.:

```
extern int array[5];
int array[10];        /* oops -- has it 5 or 10 elements? */
```

(252) argument * conflicts with prototype ***(Parser)***

The argument specified (argument 0 is the left most argument) of this function definition does not agree with a previous prototype for this function, e.g.:

```
/* this is supposedly calc's prototype */
extern int calc(int, int);
int calc(int a, long int b)  /* hmmm -- which is right? */
{                            /* error flagged here */
  return sin(b/a);
}
```

(253) argument list conflicts with prototype ***(Parser)***

The argument list in a function definition is not the same as a previous prototype for that function. Check that the number and types of the arguments are all the same.

```
extern int calc(int);   /* this is supposedly calc's prototype */
int calc(int a, int b)  /* hmmm -- which is right? */
{                       /* error flagged here */
  return a + b;
}
```

(254) undefined *: "*" ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(255) not a member of the struct/union "*" ***(Parser)***

This identifier is not a member of the structure or union type with which it used here, e.g.:

```
struct {
  int a, b, c;
} data;
if(data.d)    /* oops --
                 there is no member d in this structure */
  return;
```

(256) too much indirection ***(Parser)***

A pointer declaration may only have 16 levels of indirection.

(257) only "register" storage class allowed ***(Parser)***

The only storage class allowed for a function parameter is `register`, e.g.:

```
void process(static int input)
```

(258) duplicate qualifier ***(Parser)***

There are two occurrences of the same qualifier in this type specification. This can occur either directly or through the use of a typedef. Remove the redundant qualifier. For example:

```
typedef volatile int vint;
/* oops -- this results in two volatile qualifiers */
volatile vint very_vol;
```

(259) can't be qualifed both far and near ***(Parser)***

It is illegal to qualify a type as both `far` and `near`, e.g.:

```
far near int spooky;  /* oops -- choose far or near, not both */
```

(260) undefined enum tag "*" ***(Parser)***

This enum tag has not been defined, e.g.:

```
enum WHAT what;  /* a definition for WHAT was never seen */
```

(261) struct/union member "*" redefined ***(Parser)***

This name of this member of the struct or union has already been used in this `struct` or `union`, e.g.:

```
struct {
  int a;
  int b;
  int a;  /* oops -- a different name is required here */
} input;
```

(262) struct/union "*" redefined ***(Parser)***

A structure or union has been defined more than once, e.g.:

```
struct {
  int a;
} ms;
struct {
  int a;
} ms;    /* was this meant to be the same name as above? */
```

(263) members can't be functions ***(Parser)***

A member of a structure or a union may not be a function. It may be a pointer to a function, e.g.:

```
struct {
  int a;
  int get(int);  /* should be a pointer: int (*get)(int); */
} object;
```

(264) bad bitfield type ***(Parser)***

A bitfield may only have a type of `int` (`signed` or `unsigned`), e.g.:

```
struct FREG {
  char b0:1;   /* these must be part of an int, not char */
  char   :6;
  char b7:1;
} freg;
```

(265) integer constant expected ***(Parser)***

A *colon* appearing after a member name in a structure declaration indicates that the member is a bitfield. An integral constant must appear after the *colon* to define the number of bits in the bitfield, e.g.:

```
struct {
  unsigned first:  /* oops -- should be: unsigned first; */
  unsigned second;
} my_struct;
```

If this was meant to be a structure with bitfields, then the following illustrates an example:

```
struct {
  unsigned first : 4;  /* 4 bits wide */
  unsigned second: 4;  /* another 4 bits */
} my_struct;
```

(266) storage class illegal ***(Parser)***

A structure or union member may not be given a storage class. Its storage class is determined by the storage class of the structure, e.g.:

```
struct {
  /* no additional qualifiers may be present with members */
  static int first;
} ;
```

(267) bad storage class ***(Code Generator)***

The code generator has encountered a variable definition whose storage class is invalid, e.g.:

```
auto int foo; /* auto not permitted with global variables */
int power(static int a)  /* parameters may not be static */
{
```

```
  return foo * a;
}
```

(268) inconsistent storage class ***(Parser)***

A declaration has conflicting storage classes. Only one storage class should appear in a declaration, e.g.:

```
extern static int where;  /* so is it static or extern? */
```

(269) inconsistent type ***(Parser)***

Only one basic type may appear in a declaration, e.g.:

```
int float if;  /* is it int or float? */
```

(270) variable can’t have storage class "register" ***(Parser)***

Only function parameters or `auto` variables may be declared using the `register` qualifier, e.g.:

```
register int gi;      /* this cannot be qualified register */
int process(register int input)  /* this is okay */
{
  return input + gi;
}
```

(271) type can’t be long ***(Parser)***

Only `int` and `float` can be qualified with `long`.

```
long char lc;  /* what? */
```

(272) type can’t be short ***(Parser)***

Only `int` can be modified with `short`, e.g.:

```
short float sf;  /* what? */
```

(273) type can't be both signed and unsigned *(Parser)*

The type modifiers `signed` and `unsigned` cannot be used together in the same declaration, as they have opposite meaning, e.g.:

```
signed unsigned int confused;  /* which is it? */
```

(274) type can't be unsigned *(Parser)*

A floating point type cannot be made `unsigned`, e.g.:

```
unsigned float uf;  /* what? */
```

(275) "..." illegal in non-prototype argument list *(Parser)*

The *ellipsis* symbol may only appear as the last item in a prototyped argument list. It may not appear on its own, nor may it appear after argument names that do not have types, i.e. K&R-style non-prototype function definitions. For example:

```
/* K&R-style non-prototyped function definition */
int kandr(a, b, ...)
  int a, b;
{
```

(276) type specifier required for prototyped argument *(Parser)*

A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

(277) can't mix prototyped and non-prototyped arguments *(Parser)*

A function declaration can only have all prototyped arguments (i.e. with types inside the parentheses) or all K&R style args (i.e. only names inside the parentheses and the argument types in a declaration list before the start of the function body), e.g.:

```
int plus(int a, b)  /* oops -- a is prototyped, b is not */
int b;
{
  return a + b;
}
```

(278) argument "*" redeclared ***(Parser)***

The specified argument is declared more than once in the same argument list, e.g.

```
/* can't have two parameters called "a" */
int calc(int a, int a)
```

(279) initialization of function arguments is illegal ***(Parser)***

A function argument can't have an initialiser in a declaration. The initialisation of the argument happens when the function is called and a value is provided for the argument by the calling function, e.g.:

```
/* oops -- a is initialized when proc is called */
extern int proc(int a = 9);
```

(280) arrays of functions are illegal ***(Parser)***

You can't define an array of functions. You can however define an array of pointers to functions, e.g.:

```
int * farray[]();  /* oops -- should be: int (* farray[])(); */
```

(281) functions can't return functions ***(Parser)***

A function cannot return a function. It can return a function pointer. A function returning a pointer to a function could be declared like this: int (* (name()))(). Note the many parentheses that are necessary to make the parts of the declaration bind correctly.

(282) functions can't return arrays ***(Parser)***

A function can return only a scalar (simple) type or a structure. It cannot return an array.

(283) dimension required ***(Parser)***

Only the most significant (i.e. the first) dimension in a multi-dimension array may not be assigned a value. All succeeding dimensions must be present as a constant expression, e.g.:

```
/* This should be, e.g.: int arr[][7] */
int get_element(int arr[2][])
{
  return array[1][6];
}
```

(284) invalid dimension *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(285) no identifier in declaration *(Parser)*

The identifier is missing in this declaration. This error can also occur where the compiler has been confused by such things as missing closing braces, e.g.:

```
void interrupt(void)  /* what is the name of this function? */
{
}
```

(286) declarator too complex *(Parser)*

This declarator is too complex for the compiler to handle. Examine the declaration and find a way to simplify it. If the compiler finds it too complex, so will anybody maintaining the code.

(287) arrays of bits or pointers to bit are illegal *(Parser)*

It is not legal to have an array of bits, or a pointer to bit variable, e.g.:

```
bit barray[10];  /* wrong -- no bit arrays */
bit * bp;        /* wrong -- no pointers to bit variables */
```

(288) only functions may be void *(Parser)*

A variable may not be `void`. Only a function can be `void`, e.g.:

```
int a;
void b;  /* this makes no sense */
```

(289) only functions may be qualified "interrupt" *(Parser)*

The qualifier `interrupt` may not be applied to anything except a function, e.g.:

```
/* variables cannot be qualified interrupt */
interrupt int input;
```

(290) illegal function qualifier(s) ***(Parser)***

A qualifier has been applied to a function which makes no sense in this context. Some qualifier only make sense when used with an lvalue, e.g. const or volatile. This may indicate that you have forgotten out a star * indicating that the function should return a pointer to a qualified object, e.g.

```
const char ccrv(void) /* const * char ccrv(void) perhaps? */
{                     /* error flagged here */
  return ccip;
}
```

(291) K&R identifier "*" not an argument ***(Parser)***

This identifier that has appeared in a K&R style argument declarator is not listed inside the parentheses after the function name, e.g.:

```
int process(input)
int unput;        /* oops -- that should be int input; */
{
}
```

(292) function parameter may not be a function ***(Parser)***

A function parameter may not be a function. It may be a pointer to a function, so perhaps a "*" has been omitted from the declaration.

(293) bad size in index_type() ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(294) can't allocate * bytes of memory ***(Code Generator, Hexmate)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(295) expression too complex ***(Parser)***

This expression has caused overflow of the compiler's internal stack and should be re-arranged or split into two expressions.

(296) out of memory ***(Objtohex)***

This could be an internal compiler error. Contact HI-TECH Software technical support with details.

(297) bad argument (*) to tysize() ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(298) end of file in #asm ***(Preprocessor)***

An end of file has been encountered inside a `#asm` block. This probably means the `#endasm` is missing or misspelt, e.g.:

```
#asm
  mov  r0, #55
  mov  [r1], r0
}               /* oops -- where is the #endasm */
```

(300) unexpected end of file ***(Parser)***

An end-of-file in a C module was encountered unexpectedly, e.g.:

```
void main(void)
{
  init();
  run();    /* is that it? What about the close brace */
```

(301) end of file on string file ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(302) can't reopen "*": * ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(303) can't allocate * bytes of memory (line *) ***(Parser)***

The parser was unable to allocate memory for the longest string encountered, as it attempts to sort and merge strings. Try reducing the number or length of strings in this module.

(306) can't allocate * bytes of memory for * ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(307) too many qualifier names *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(308) too many case labels in switch *(Code Generator)*

There are too many `case` labels in this `switch` statement. The maximum allowable number of `case` labels in any one `switch` statement is 511.

(309) too many symbols *(Assembler)*

There are too many symbols for the assembler's symbol table. Reduce the number of symbols in your program.

(310) "]" expected *(Parser)*

A closing square bracket was expected in an array declaration or an expression using an array index, e.g.

```
process(carray[idx);  /* oops --
                         should be: process(carray[idx]); */
```

(311) closing quote expected *(Parser)*

A closing quote was expected for the indicated string.

(312) "*" expected *(Parser)*

The indicated token was expected by the parser.

(313) function body expected *(Parser)*

Where a function declaration is encountered with K&R style arguments (i.e. argument names but no types inside the parentheses) a function body is expected to follow, e.g.:

```
/* the function block must follow, not a semicolon */
int get_value(a, b);
```

(314) ";" expected *(Parser)*

A *semicolon* is missing from a statement. A close brace or keyword was found following a statement with no terminating *semicolon*, e.g.:

```
while(a) {
  b = a--  /* oops -- where is the semicolon? */
}          /* error is flagged here */
```

Note: Omitting a semicolon from statements not preceding a close brace or keyword typically results in some other error being issued for the following code which the parser assumes to be part of the original statement.

(315) "{" expected *(Parser)*

An *opening brace* was expected here. This error may be the result of a function definition missing the *opening brace*, e.g.:

```
/* oops! no opening brace after the prototype */
void process(char c)
  return max(c, 10) * 2; /* error flagged here */
}
```

(316) "}" expected *(Parser)*

A *closing brace* was expected here. This error may be the result of a initialized array missing the *closing brace*, e.g.:

```
char carray[4] = { 1, 2, 3, 4;  /* oops -- no closing brace */
```

(317) "(" expected *(Parser)*

An *opening parenthesis*, `(`, was expected here. This must be the first token after a `while`, `for`, `if`, `do` or `asm` keyword, e.g.:

```
if a == b  /* should be: if(a == b) */
  b = 0;
```

(318) string expected *(Parser)*

The operand to an `asm` statement must be a string enclosed in parentheses, e.g.:

```
asm(nop);  /* that should be asm("nop");
```

(319) while expected ***(Parser)***

The keyword `while` is expected at the end of a `do` statement, e.g.:

```
do {
  func(i++);
}             /* do the block while what condition is true? */
if(i > 5)     /* error flagged here */
  end();
```

(320) ":" expected ***(Parser)***

A *colon* is missing after a `case` label, or after the keyword `default`. This often occurs when a *semicolon* is accidentally typed instead of a *colon*, e.g.:

```
switch(input) {
  case 0;         /* oops -- that should have been: case 0: */
    state = NEW;
```

(321) label identifier expected ***(Parser)***

An identifier denoting a label must appear after `goto`, e.g.:

```
if(a)
  goto 20;
/* this is not BASIC -- a valid C label must follow a goto */
```

(322) enum tag or "{" expected ***(Parser)***

After the keyword `enum` must come either an identifier that is or will be defined as an `enum` tag, or an opening brace, e.g.:

```
enum 1, 2;  /* should be, e.g.: enum {one=1, two }; */
```

(323) struct/union tag or "{" expected ***(Parser)***

An identifier denoting a structure or union or an opening brace must follow a `struct` or `union` keyword, e.g.:

```
struct int a;  /* this is not how you define a structure */
```

You might mean something like:

```
struct {
  int a;
} my_struct;
```

(324) too many arguments for printf-style format string ***(Parser)***

There are too many arguments for this format string. This is harmless, but may represent an incorrect format string, e.g.:

```
/* oops -- missed a placeholder? */
printf("%d - %d", low, high, median);
```

(325) error in printf-style format string ***(Parser)***

There is an error in the format string here. The string has been interpreted as a `printf()` style format string, and it is not syntactically correct. If not corrected, this will cause unexpected behaviour at run time, e.g.:

```
printf("%l", lll);  /* oops -- maybe: printf("%ld", lll); */
```

(326) long int argument required in printf-style format string ***(Parser)***

A long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%lx", 2);  // maybe you meant: printf("%lx", 2L);
```

(327) long long int argument required in printf-style format string ***(Parser)***

A long long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%llx", 2);  // maybe you meant: printf("%llx", 2LL);
```

Note that not all HI-TECH C compilers provide support for a long long integer type.

(328) int argument required in printf-style format string ***(Parser)***

An integral argument is required for this printf-style format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%d", 1.23); /* wrong number or wrong placeholder */
```

(329) double argument required in printf-style format string ***(Parser)***

The printf format specifier corresponding to this argument is `%f` or similar, and requires a floating point expression. Check for missing or extra format specifiers or arguments to printf.

```
printf("%f", 44);  /* should be: printf("%f", 44.0); */
```

(330) pointer to * argument required in printf-style format string ***(Parser)***

A pointer argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

(331) too few arguments for printf-style format string ***(Parser)***

There are too few arguments for this format string. This would result in a garbage value being printed or converted at run time, e.g.:

```
printf("%d - %d", low);
  /* oops! where is the other value to print? */
```

(332) "interrupt_level" should be 0 to 7 ***(Parser)***

The pragma `interrupt_level` must have an argument from 0 to 7, e.g.:

```
#pragma interrupt_level  /* oops -- what is the level */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(333) unrecognized qualifier name after "strings" ***(Parser)***

The `pragma strings` was passed a qualifier that was not identified, e.g.:

```
/* oops -- should that be #pragma strings const ? */
#pragma strings cinst
```

(334) unrecognized qualifier name after "printf_check" ***(Parser)***

The `#pragma printf_check` was passed a qualifier that could not be identified, e.g.:

```
/* oops -- should that be const not cinst? */
#pragma printf_check(printf) cinst
```

(335) unknown pragma "*" *(Parser)*

An unknown `pragma` directive was encountered, e.g.:

```
#pragma rugsused w  /* I think you meant regsused */
```

(336) string concatenation across lines *(Parser)*

Strings on two lines will be concatenated. Check that this is the desired result, e.g.:

```
char * cp = "hi"
  "there";    /* this is okay,
                 but is it what you had intended? */
```

(337) line does not have a newline on the end *(Parser)*

The last line in the file is missing the *newline* (operating system dependent character) from the end. Some editors will create such files, which can cause problems for include files. The ANSI C standard requires all source files to consist of complete lines only.

(338) can't create * file "*" *(Any)*

The application tried to create or open the named file, but it could not be created. Check that all file pathnames are correct.

(339) initializer in extern declaration *(Parser)*

A declaration containing the keyword `extern` has an initialiser. This overrides the `extern` storage class, since to initialise an object it is necessary to define (i.e. allocate storage for) it, e.g.:

```
extern int other = 99;  /* if it's extern and not allocated
                        storage, how can it be initialized? */
```

(340) string not terminated by null character. *(Parser)*

A char array is being initialized with a string literal larger than the array. Hence there is insufficient space in the array to safely append a null terminating character, e.g.:

```
char foo[5] = "12345"; /* the string stored in foo won't have
                          a null terminating, i.e.
                          foo = ['1', '2', '3', '4', '5'] */
```

(343) implicit return at end of non-void function *(Parser)*

A function which has been declared to return a value has an execution path that will allow it to reach the end of the function body, thus returning without a value. Either insert a `return` statement with a value, or if the function is not to return a value, declare it `void`, e.g.:

```
int mydiv(double a, int b)
{
  if(b != 0)
    return a/b;   /* what about when b is 0? */
}                 /* warning flagged here */
```

(344) non-void function returns no value *(Parser)*

A function that is declared as returning a value has a `return` statement that does not specify a return value, e.g.:

```
int get_value(void)
{
  if(flag)
    return val++;
  return;
  /* what is the return value in this instance? */
}
```

(345) unreachable code *(Parser)*

This section of code will never be executed, because there is no execution path by which it could be reached, e.g.:

```
while(1)          /* how does this loop finish? */
  process();
flag = FINISHED;  /* how do we get here? */
```

(346) declaration of "*" hides outer declaration *(Parser)*

An object has been declared that has the same name as an outer declaration (i.e. one outside and preceding the current function or block). This is legal, but can lead to accidental use of one variable when the outer one was intended, e.g.:

```
int input;          /* input has filescope */
void process(int a)
{
  int input;        /* local blockscope input */
  a = input;        /* this will use the local variable.
                       Is this right? */
```

(347) external declaration inside function ***(Parser)***

A function contains an `extern` declaration. This is legal but is invariably not desirable as it restricts the scope of the function declaration to the function body. This means that if the compiler encounters another declaration, use or definition of the extern object later in the same file, it will no longer have the earlier declaration and thus will be unable to check that the declarations are consistent. This can lead to strange behaviour of your program or signature errors at link time. It will also hide any previous declarations of the same thing, again subverting the compiler's type checking. As a general rule, always declare `extern` variables and functions outside any other functions. For example:

```
int process(int a)
{
  /* this would be better outside the function */
  extern int away;
  return away + a;
}
```

(348) auto variable "*" should not be qualified ***(Parser)***

An `auto` variable should not have qualifiers such as `near` or `far` associated with it. Its storage class is implicitly defined by the stack organization. An `auto` variable may be qualified with `static`, but it is then no longer `auto`.

(349) non-prototyped function declaration for "*" ***(Parser)***

A function has been declared using old-style (K&R) arguments. It is preferable to use prototype declarations for all functions, e.g.:

```
int process(input)
int input;          /* warning flagged here */
{
}
```

This would be better written:

```
int process(int input)
{
}
```

(350) unused * "*" (from line *) ***(Parser)***

The indicated object was never used in the function or module being compiled. Either this object is redundant, or the code that was meant to use it was excluded from compilation or misspelt the name of the object. Note that the symbols `rcsid` and `sccsid` are never reported as being unused.

(352) float parameter coerced to double ***(Parser)***

Where a non-prototyped function has a parameter declared as `float`, the compiler converts this into a `double float`. This is because the default C type conversion conventions provide that when a floating point number is passed to a non-prototyped function, it will be converted to `double`. It is important that the function declaration be consistent with this convention, e.g.:

```
double inc_flt(f)  /* f will be converted to double */
float f;           /* warning flagged here */
{
  return f * 2;
}
```

(353) sizeof external array "*" is zero ***(Parser)***

The size of an external array evaluates to zero. This is probably due to the array not having an explicit dimension in the extern declaration.

(354) possible pointer truncation ***(Parser)***

A pointer qualified far has been assigned to a default pointer or a pointer qualified near, or a default pointer has been assigned to a pointer qualified near. This may result in truncation of the pointer and loss of information, depending on the memory model in use.

(355) implicit signed to unsigned conversion ***(Parser)***

A `signed` number is being assigned or otherwise converted to a larger `unsigned` type. Under the ANSI "value preserving" rules, this will result in the `signed` value being first sign-extended to a `signed` number the size of the target type, then converted to `unsigned` (which involves no change in bit pattern). Thus an unexpected sign extension can occur. To ensure this does not happen, first convert the signed value to an unsigned equivalent, e.g.:

```
signed char sc;
unsigned int ui;
ui = sc;    /* if sc contains 0xff,
               ui will contain 0xffff for example */
```

will perform a sign extension of the `char` variable to the longer type. If you do not want this to take place, use a cast, e.g.:

```
ui = (unsigned char)sc;
```

(356) implicit conversion of float to integer *(Parser)*

A floating point value has been assigned or otherwise converted to an integral type. This could result in truncation of the floating point value. A typecast will make this warning go away.

```
double dd;
int i;
i = dd;    /* is this really what you meant? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)dd;
```

(357) illegal conversion of integer to pointer *(Parser)*

An integer has been assigned to or otherwise converted to a pointer type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This may also mean you have forgotten the `&` address operator, e.g.:

```
int * ip;
int i;
ip = i;    /* oops -- did you mean ip = &i ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
ip = (int *)i;
```

(358) illegal conversion of pointer to integer ***(Parser)***

A pointer has been assigned to or otherwise converted to a integral type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This may also mean you have forgotten the * dereference operator, e.g.:

```
int * ip;
int i;
i = ip;    /* oops -- did you mean i = *ip ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)ip;
```

(359) illegal conversion between pointer types ***(Parser)***

A pointer of one type (i.e. pointing to a particular kind of object) has been converted into a pointer of a different type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed, e.g.:

```
long input;
char * cp;
cp = &input;  /* is this correct? */
```

This is common way of accessing bytes within a multi-byte variable. To indicate that this is the intended operation of the program, use a cast:

```
cp = (char *)&input;  /* that's better */
```

This warning may also occur when converting between pointers to objects which have the same type, but which have different qualifiers, e.g.:

```
char * cp;
/* yes, but what sort of characters? */
cp = "I am a string of characters";
```

If the default type for string literals is `const char *`, then this warning is quite valid. This should be written:

```
const char * cp;
cp = "I am a string of characters";  /* that's better */
```

Omitting a qualifier from a pointer type is often disastrous, but almost certainly not what you intend.

(360) array index out of bounds *(Parser)*

An array is being indexed with a constant value that is less than zero, or greater than or equal to the number of elements in the array. This warning will not be issued when accessing an array element via a pointer variable, e.g.:

```
int i, * ip, input[10];
i = input[-2];           /* oops -- this element doesn't exist */
ip = &input[5];
i = ip[-2];              /* this is okay */
```

(361) function declared implicit int *(Parser)*

Where the compiler encounters a function call of a function whose name is presently undefined, the compiler will automatically declare the function to be of type `int`, with unspecified (K&R style) parameters. If a definition of the function is subsequently encountered, it is possible that its type and arguments will be different from the earlier implicit declaration, causing a compiler error. The solution is to ensure that all functions are defined or at least declared before use, preferably with prototyped parameters. If it is necessary to make a forward declaration of a function, it should be preceded with the keywords `extern` or `static` as appropriate. For example:

```
/* I may prevent an error arising from calls below */
void set(long a, int b);
void main(void)
{
  /* by here a prototype for set should have seen */
  set(10L, 6);
}
```

(362) redundant "&" applied to array *(Parser)*

The address operator `&` has been applied to an array. Since using the name of an array gives its address anyway, this is unnecessary and has been ignored, e.g.:

```
int array[5];
int * ip;
/* array is a constant, not a variable; the & is redundant. */
ip = &array;
```

(363) redundant "&" or "*" applied to function address *(Parser)*

The address operator "&" has been applied to a function. Since using the name of a function gives its address anyway, this is unnecessary and has been ignored, e.g.:

```
extern void foo(void);
void main(void)
{
    void(*bar)(void);
    /* both assignments are equivalent */
    bar = &foo;
    bar = foo;  /* the & is redundant */
}
```

(364) attempt to modify object qualified * *(Parser)*

Objects declared `const` or `code` may not be assigned to or modified in any other way by your program. The effect of attempting to modify such an object is compiler-specific.

```
const int out = 1234;  /* "out" is read only */
out = 0;               /* oops --
                          writing to a read-only object */
```

(365) pointer to non-static object returned *(Parser)*

This function returns a pointer to a non-`static` (e.g. `auto`) variable. This is likely to be an error, since the storage associated with automatic variables becomes invalid when the function returns, e.g.:

```
char * get_addr(void)
{
  char c;
  /* returning this is dangerous;
     the pointer could be dereferenced */
  return &c;
}
```

(366) operands of "*" not same pointer type *(Parser)*

The operands of this operator are of different pointer types. This probably means you have used the wrong pointer, but if the code is actually what you intended, use a typecast to suppress the error message.

(367) identifier is already extern; can't be static *(Parser)*

This function was already declared `extern`, possibly through an implicit declaration. It has now been redeclared `static`, but this redeclaration is invalid.

```
void main(void)
{
  /* at this point the compiler assumes set is extern... */
  set(10L, 6);
}
/* now it finds out otherwise */
static void set(long a, int b)
{
  PORTA = a + b;
}
```

(368) array dimension on "*[]" ignored *(Preprocessor)*

An array dimension on a function parameter has been ignored because the argument is actually converted to a pointer when passed. Thus arrays of any size may be passed. Either remove the dimension from the parameter, or define the parameter using pointer syntax, e.g.:

```
/* param should be: "int array[]" or "int *" */
int get_first(int array[10])
{                              /* warning flagged here */
  return array[0];
}
```

(369) signed bitfields not supported *(Parser)*

Only `unsigned` bitfields are supported. If a bitfield is declared to be type `int`, the compiler still treats it as `unsigned`, e.g.:

```
struct {
  signed int sign: 1;    /* this must be unsigned */
  signed int value: 15;
} ;
```

(370) illegal basic type; int assumed *(Parser)*

The basic type of a cast to a qualified basic type couldn't not be recognised and the basic type was assumed to be `int`, e.g.:

```
/* here ling is assumed to be int */
unsigned char bar = (unsigned ling) 'a';
```

(371) missing basic type; int assumed ***(Parser)***

This declaration does not include a basic type, so `int` has been assumed. This declaration is not illegal, but it is preferable to include a basic type to make it clear what is intended, e.g.:

```
char c;
i;      /* don't let the compiler make assumptions, use : int i */
func(); /* ditto, use: extern int func(int); */
```

(372) "," expected ***(Parser)***

A *comma* was expected here. This could mean you have left out the *comma* between two identifiers in a declaration list. It may also mean that the immediately preceding type name is misspelled, and has thus been interpreted as an identifier, e.g.:

```
unsigned char a;
/* thinks: chat & b are unsigned, but where is the comma? */
unsigned chat b;
```

(373) implicit signed to unsigned conversion ***(Parser)***

An `unsigned` type was expected where a `signed` type was given and was implicitly cast to `unsigned`, e.g.:

```
unsigned int foo = -1;
/* the above initialization is implicitly treated as:
   unsigned int foo = (unsigned) -1; */
```

(374) missing basic type; int assumed ***(Parser)***

The basic type of a cast to a qualified basic type was missing and assumed to be `int`., e.g.:

```
int i = (signed) 2; /* (signed) assumed to be (signed int) */
```

(375) unknown FNREC type "*" ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(376) bad non-zero node in call graph ***(Linker)***

The linker has encountered a top level node in the call graph that is referenced from lower down in the call graph. This probably means the program has indirect recursion, which is not allowed when using a compiled stack.

(378) can't create * file "*" ***(Hexmate)***

This type of file could not be created. Is the file or a file by this name already in use?

(379) bad record type "*" ***(Linker)***

This is an internal compiler error. Ensure the object file is a valid HI-TECH object file. Contact HI-TECH Software technical support with details.

(380) unknown record type (*) ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(381) record "*" too long (*) ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(382) incomplete record: type = *, length = * ***(Dump, Xstrip)***

This message is produced by the DUMP or XSTRIP utilities and indicates that the object file is not a valid HI-TECH object file, or that it has been truncated. Contact HI-TECH Support with details.

(383) text record has length (*) too small ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(384) assertion failed: file *, line *, expression * ***(Linker, Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(387) illegal or too many -G options ***(Linker)***

There has been more than one linker `-g` option, or the `-g` option did not have any arguments following. The arguments specify how the segment addresses are calculated.

(388) duplicate -M option ***(Linker)***

The map file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of this option is present on the command line. See Section 5.7.9 for information on the correct syntax for this option.

(389) illegal or too many -O options ***(Linker)***

This linker `-o` flag is illegal, or another `-o` option has been encountered. A `-o` option to the linker must be immediately followed by a filename with no intervening space.

(390) missing argument to -P ***(Linker)***

There have been too many `-p` options passed to the linker, or a `-p` option was not followed by any arguments. The arguments of separate `-p` options may be combined and separated by *commas*.

(391) missing argument to -Q ***(Linker)***

The `-Q` linker option requires the machine type for an argument.

(392) missing argument to -U ***(Linker)***

The `-U` (undefine) option needs an argument.

(393) missing argument to -W ***(Linker)***

The `-W` option (listing width) needs a numeric argument.

(394) duplicate -D or -H option ***(Linker)***

The symbol file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of either of these options is present on the command line.

(395) missing argument to -J ***(Linker)***

The maximum number of errors before aborting must be specified following the `-j` linker option.

(397) usage: hlink [-options] files.obj files.lib ***(Linker)***

Improper usage of the command-line linker. If you are invoking the linker directly then please refer to Section 5.7 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(398) output file can't be also an input file ***(Linker)***

The linker has detected an attempt to write its output file over one of its input files. This cannot be done, because it needs to simultaneously read and write input and output files.

(400) bad object code format ***(Linker)***

This is an internal compiler error. The object code format of an object file is invalid. Ensure it is a valid HI-TECH object file. Contact HI-TECH Software technical support with details.

(402) bad argument to -F ***(Objtohex)***

The `-F` option for `objtohex` has been supplied an invalid argument. If you are invoking this command-line tool directly then please refer to Section 5.11 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(403) bad -E option: "*" ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(404) bad maximum length value to -<digits> ***(Objtohex)***

The first value to the OBJTOHEX `-n,m` hex length/rounding option is invalid.

(405) bad record size rounding value to -<digits> ***(Objtohex)***

The second value to the OBJTOHEX `-n,m` hex length/rounding option is invalid.

(406) bad argument to -A ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(407) bad argument to -U ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(408) bad argument to -B *(Objtohex)*

This option requires an integer argument in either base 8, 10 or 16. If you are invoking `objtohex` directly then see Section 5.11 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(409) bad argument to -P *(Objtohex)*

This option requires an integer argument in either base 8, 10 or 16. If you are invoking `objtohex` directly then see Section 5.11 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(410) bad combination of options *(Objtohex)*

The combination of options supplied to `OBJTOHEX` is invalid.

(412) text does not start at 0 *(Objtohex)*

Code in some things must start at zero. Here it doesn't.

(413) write error on "*" *(Assembler, Linker, Cromwell)*

A write error occurred on the named file. This probably means you have run out of disk space.

(414) read error on "*" *(Linker)*

The linker encountered an error trying to read this file.

(415) text offset too low in COFF file *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(416) bad character (*) in extended TEKHEX line *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(417) seek error in "*" *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(418) image too big ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(419) object file is not absolute ***(Objtohex)***

The object file passed to `OBJTOHEX` has relocation items in it. This may indicate it is the wrong object file, or that the linker or OBJTOHEX have been given invalid options. The object output files from the assembler are relocatable, not absolute. The object file output of the linker is absolute.

(420) too many relocation items ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(421) too many segments ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(422) no end record ***(Linker)***

This object file has no end record. This probably means it is not an object file. Contact HI-TECH Support if the object file was generated by the compiler.

(423) illegal record type ***(Linker)***

There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Contact HI-TECH Support with details if the object file was created by the compiler.

(424) record too long ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(425) incomplete record ***(Objtohex, Libr)***

The object file passed to OBJTOHEX or the librarian is corrupted. Contact HI-TECH Support with details.

(427) syntax error in checksum list ***(Objtohex)***

There is a syntax error in a checksum list read by OBJTOHEX. The checksum list is read from standard input in response to an option.

(428) too many segment fixups ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(429) bad segment fixups ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(430) bad checksum specification ***(Objtohex)***

A checksum list supplied to `OBJTOHEX` is syntactically incorrect.

(431) bad argument to -E ***(Objtoexe)***

This option requires an integer argument in either base 8, 10 or 16. If you are invoking `objtoexe` directly then check this argument. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(432) usage: objtohex [-ssymfile] [object-file [exe-file]] ***(Objtohex)***

Improper usage of the command-line tool `objtohex`. If you are invoking `objtohex` directly then please refer to Section 5.11 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(434) too many symbols (*) ***(Linker)***

There are too many symbols in the symbol table, which has a limit of * symbols. Change some global symbols to local symbols to reduce the number of symbols.

(435) bad segment selector "*" ***(Linker)***

The segment specification option (`-G`) to the linker is invalid, e.g.:

```
-GA/f0+10
```

Did you forget the radix?

```
-GA/f0h+10
```

(436) psect "*" re-orged ***(Linker)***

This psect has had its start address specified more than once.

(437) missing "=" in class spec *(Linker)*

A class spec needs an = sign, e.g. -Ctext=ROM See Section 5.7.9 for more information.

(438) bad size in -S option *(Linker)*

The address given in a `-S` specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing `O`, for octal, or `H` for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, e.g.:

```
-SCODE=f000
```

Did you forget the radix?

```
-SCODE=f000h
```

(439) bad -D spec: "*" *(Linker)*

The format of a `-D` specification, giving a *delta* value to a class, is invalid, e.g.:

```
-DCODE
```

What is the *delta* value for this class? Maybe you meant something like:

```
-DCODE=2
```

(440) bad delta value in -D spec *(Linker)*

The *delta* value supplied to a `-D` specification is invalid. This value should an integer of base 8, 10 or 16.

(441) bad -A spec: "*" *(Linker)*

The format of a `-A` specification, giving address ranges to the linker, is invalid, e.g.:

```
-ACODE
```

What is the range for this class? Maybe you meant:

```
-ACODE=0h-1fffh
```

(442) missing address in -A spec *(Linker)*

The format of a `-A` specification, giving address ranges to the linker, is invalid, e.g.:

```
-ACODE=
```

What is the range for this class? Maybe you meant:

```
-ACODE=0h-1fffh
```

(443) bad low address "*" in -A spec *(Linker)*

The low address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-ACODE=1fff-3fffh
```

Did you forget the radix?

```
-ACODE=1fffh-3fffh
```

(444) expected "-" in -A spec *(Linker)*

There should be a minus sign, `-`, between the high and low addresses in a `-A` linker option, e.g.

```
-AROM=1000h
```

maybe you meant:

```
-AROM=1000h-1fffh
```

(445) bad high address "*" in -A spec *(Linker)*

The high address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing `O`, for octal, or `H` for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, e.g.:

```
-ACODE=0h-ffff
```

Did you forget the radix?

```
-ACODE=0h-ffffh
```

See Section 5.7.20 for more information.

(446) bad overrun address "*" in -A spec ***(Linker)***

The overrun address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-AENTRY=0-0FFh-1FF
```

Did you forget the radix?

```
-AENTRY=0-0FFh-1FFh
```

(447) bad load address "*" in -A spec ***(Linker)***

The load address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing `O` (for octal) or `H` for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-ACODE=0h-3fffh/a000
```

Did you forget the radix?

```
-ACODE=0h-3fffh/a000h
```

(448) bad repeat count "*" in -A spec ***(Linker)***

The repeat count given in a `-A` specification is invalid, e.g.:

```
-AENTRY=0-0FFhxf
```

Did you forget the radix?

```
-AENTRY=0-0FFhxfh
```

(449) syntax error in -A spec: * ***(Linker)***

The `-A` spec is invalid. A valid -A spec should be something like:
-AROM=1000h-1FFFh

(450) psect "*" was never defined ***(Linker, Optimiser)***

This psect has been listed in a `-P` option, but is not defined in any module within the program.

(451) bad psect origin format in -P option ***(Linker)***

The origin format in a `-p` option is not a validly formed decimal, octal or hex number, nor is it the name of an existing psect. A hex number must have a trailing H, e.g.:

```
-pbss=f000
```

Did you forget the radix?

```
-pbss=f000h
```

(452) bad "+" (minimum address) format in -P option ***(Linker)***

The minimum address specification in the linker's `-p` option is badly formatted, e.g.:

```
-pbss=data+f000
```

Did you forget the radix?

```
-pbss=data+f000h
```

(453) missing number after "%" in -P option ***(Linker)***

The `%` operator in a `-p` option (for rounding boundaries) must have a number after it.

(454) link and load address can't both be set to "." in -P option ***(Linker)***

The link and load address of a psect have both been specified with a *dot* character. Only one of these addresses may be specified in this manner, e.g.:

```
-Pmypsect=1000h/.
-Pmypsect=./1000h
```

Both of these options are valid and equivalent, however the following usage is ambiguous:

```
-Pmypsect=./.
```

What is the link or load address of this psect?

(455) psect "*" not relocated on 0x* byte boundary ***(Linker)***

This psect is not relocated on the required boundary. Check the relocatability of the psect and correct the `-p` option. if necessary.

(456) psect "*" not loaded on 0x* boundary ***(Linker)***

This psect has a relocatability requirement that is not met by the load address given in a `-p` option. For example if a psect must be on a 4K byte boundary, you could not start it at 100H.

(459) remove failed, error: *, * ***(xstrip)***

The creation of the output file failed when removing an interemediate file.

(460) rename failed, error: *, * ***(xstrip)***

The creation of the output file failed when renaming an interemediate file.

(461) can't create * file "*" ***(Assembler, Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(464) missing key in avmap file ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(465) undefined symbol "*" in FNBREAK record ***(Linker)***

The linker has found an undefined symbol in the `FNBREAK` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(466) undefined symbol "*" in FNINDIR record ***(Linker)***

The linker has found an undefined symbol in the `FNINDIR` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(467) undefined symbol "*" in FNADDR record ***(Linker)***

The linker has found an undefined symbol in the FNADDR record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(468) undefined symbol "*" in FNCALL record ***(Linker)***

The linker has found an undefined symbol in the `FNCALL` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(469) undefined symbol "*" in FNROOT record ***(Linker)***

The linker has found an undefined symbol in the `FNROOT` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(470) undefined symbol "*" in FNSIZE record ***(Linker)***

The linker has found an undefined symbol in the `FNSIZE` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(471) recursive function calls: ***(Linker)***

These functions (or function) call each other recursively. One or more of these functions has statically allocated local variables (compiled stack). Either use the `reentrant` keyword (if supported with this compiler) or recode to avoid recursion, e.g.:

```
int test(int a)
{
   if(a == 5) {
      /* recursion may not be supported by some compilers */
      return test(a++);
   }
   return 0;
}
```

(472) non-reentrant function "*" appears in multiple call graphs: rooted at "*" and "*" ***(Linker)***

This function can be called from both main-line code and interrupt code. Use the `reentrant` keyword, if this compiler supports it, or recode to avoid using local variables or parameters, or duplicate the function, e.g.:

```
void interrupt my_isr(void)
{
  scan(6);     /* scan is called from an interrupt function */
}
```

```
void process(int a)
{
  scan(a);     /* scan is also called from main-line code */
}
```

(473) function "*" is not called from specified interrupt_level ***(Linker)***

The indicated function is never called from an interrupt function of the same interrupt level, e.g.:

```
#pragma interrupt_level 1
void foo(void)
{
    ...
}
#pragma interrupt_level 1
void interrupt bar(void)
{
    // this function never calls foo()
}
```

(474) no psect specified for function variable/argument allocation ***(Linker)***

The `FNCONF` assembler directive which specifies to the linker information regarding the auto/parameter block was never seen. This is supplied in the standard runtime files if necessary. This error may imply that the correct run-time startup module was not linked. Ensure you have used the `FNCONF` directive if the runtime startup module is hand-written.

(475) conflicting FNCONF records ***(Linker)***

The linker has seen two conflicting `FNCONF` directives. This directive should only be specified once and is included in the standard runtime startup code which is normally linked into every program.

(476) fixup overflow referencing * * (location 0x* (0x*+*), size *, value 0x*) ***(Linker)***

The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. See the following error message (477) for more information..

(477) fixup overflow in expression (location 0x* (0x*+*), size *, value 0x*) ***(Linker)***

Fixup is the process conducted by the linker of replacing symbolic references to variables etc, in an assembler instruction with an absolute value. This takes place after positioning the psects (program

sections or blocks) into the available memory on the target device. Fixup overflow is when the value determined for a symbol is too large to fit within the allocated space within the assembler instruction. For example, if an assembler instruction has an 8-bit field to hold an address and the linker determines that the symbol that has been used to represent this address has the value 0x110, then clearly this value cannot be inserted into the instruction.

The causes for this can be many, but hand-written assembler code is always the first suspect. Badly written C code can also generate assembler that ultimately generates fixup overflow errors. Consider the following error message.

```
main.obj: 8: Fixup overflow in expression (loc 0x1FD (0x1FC+1),
    size 1, value 0x7FC)
```

This indicates that the file causing the problem was `main.obj`. This would be typically be the output of compiling `main.c` or `main.as`. This tells you the file in which you should be looking. The next number (8 in this example) is the record number in the object file that was causing the problem. If you use the `DUMP` utility to examine the object file, you can identify the record, however you do not normally need to do this.

The location (`loc`) of the instruction (0x1FD), the `size` (in bytes) of the field in the instruction for the value (1) , and the `value` which is the actual value the symbol represents, is typically the only information needed to track down the cause of this error. Note that a size which is not a multiple of 8 bits will be rounded up to the nearest byte size, i.e. a 7 bit space in an instruction will be shown as 1 byte.

Generate an assembler list file for the appropriate module. Look for the address specified in the error message.

```
7    07FC    0E21  movlw 33
8    07FD     6FFC  movwf _foo
9    07FE    0012  return
```

and to confirm, look for the symbol referenced in the assembler instruction at this address in the symbol table at the bottom of the same file.

```
Symbol Table                    Fri Aug 12 13:17:37 2004
_foo 01FC   _main 07FF
```

In this example, the instruction causing the problem takes an 8-bit offset into a bank of memory, but clearly the address 0x1FC exceeds this size. Maybe the instruction should have been written as:

```
movwf  (_foo&0ffh)
```

which masks out the top bits of the address containing the bank information.

If the assembler instruction that caused this error was generated by the compiler, in the assembler list file look back up the file from the instruction at fault to determine which C statement has generated this instruction. You will then need to examine the C code for possible errors. incorrectly qualified pointers are an common trigger.

(478) * range check failed (location 0x* (0x*+*), value 0x* > limit 0x*) ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(479) circular indirect definition of symbol "*" ***(Linker)***

The specified symbol has been equated to an external symbol which, in turn, has been equated to the first symbol.

(480) function signatures do not match: * (*): 0x*/0x* ***(Linker)***

The specified function has different signatures in different modules. This means it has been declared differently, e.g. it may have been prototyped in one module and not another. Check what declarations for the function are visible in the two modules specified and make sure they are compatible, e.g.:

```
extern int get_value(int in);
/* and in another module: */
/* this is different to the declaration */
int get_value(int in, char type)
{
```

(481) common symbol "*" psect conflict ***(Linker)***

A common symbol has been defined to be in more than one psect.

(482) symbol "*" is defined more than once in "*" ***(Assembler)***

This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, e.g.:

```
_next:
  move r0, #55
  move [r1], r0
_next:          ; oops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an *underscore* prepended to their name after compilation.

(483) symbol "*" can't be global ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(484) psect "*" can't be in classes "*" and "*" ***(Linker)***

A psect cannot be in more than one class. This is either due to assembler modules with conflicting `class=` options to the PSECT directive, or use of the `-C` option to the linker, e.g.:

```
psect final,class=CODE
finish:
/* elsewhere: */
psect final,class=ENTRY
```

(485) unknown "with" psect referenced by psect "*" ***(Linker)***

The specified psect has been placed with a psect using the psect `with` flag. The psect it has been placed with does not exist, e.g.:

```
psect starttext,class=CODE,with=rext
    ; was that meant to be with text?
```

(486) psect "*" selector value redefined ***(Linker)***

The selector value for this psect has been defined more than once.

(487) psect "*" type redefined: */* ***(Linker)***

This psect has had its type defined differently by different modules. This probably means you are trying to link incompatible object modules, e.g. linking 386 flat model code with 8086 real mode code.

(488) psect "*" memory space redefined: */* ***(Linker)***

A global psect has been defined in two different memory spaces. Either rename one of the psects or, if they are the same psect, place them in the same memory space using the `space` psect flag, e.g.:

```
psect spdata,class=RAM,space=0
  ds 6
; elsewhere:
psect spdata,class=RAM,space=1
```

(489) psect "*" memory delta redefined: */* *(Linker)*

A global psect has been defined with two different delta values, e.g.:

```
psect final,class=CODE,delta=2
finish:
; elsewhere:
psect final,class=CODE,delta=1
```

(490) class "*" memory space redefined: */* *(Linker)*

A class has been defined in two different memory spaces. Either rename one of the classes or, if they are the same class, place them in the same memory space.

(491) can't find 0x* words for psect "*" in segment "*" *(Linker)*

One of the main tasks the linker performs is positioning the blocks (or psects) of code and data that is generated from the program into the memory available for the target device. This error indicates that the linker was unable to find an area of free memory large enough to accommodate one of the psects. The error message indicates the name of the psect that the linker was attempting to position and the segment name which is typically the name of a class which is defined with a linker `-A` option.

Section 3.8.1 lists each compiler-generated psect and what it contains. Typically psect names which are, or include, `text` relate to program code. Names such as `bss` or `data` refer to variable blocks. This error can be due to two reasons.

First, the size of the program or the program's data has exceeded the total amount of space on the selected device. In other words, some part of your device's memory has completely filled. If this is the case, then the size of the specified psect must be reduced.

The second cause of this message is when the total amount of memory needed by the psect being positioned is sufficient, but that this memory is fragmented in such a way that the largest contiguous block is too small to accommodate the psect. The linker is unable to split psects in this situation. That is, the linker cannot place part of a psect at one location and part somewhere else. Thus, the linker must be able to find a contiguous block of memory large enough for every psect. If this is the cause of the error, then the psect must be split into smaller psects if possible.

To find out what memory is still available, generate and look in the map file, see Section 2.6.9 for information on how to generate a map file. Search for the string `UNUSED ADDRESS RANGES`. Under

this heading, look for the name of the segment specified in the error message. If the name is not present, then all the memory available for this psect has been allocated. If it is present, there will be one address range specified under this segment for each free block of memory. Determine the size of each block and compare this with the number of words specified in the error message.

Psects containing code can be reduced by using all the compiler's optimizations, or restructuring the program. If a code psect must be split into two or more small psects, this requires splitting a function into two or more smaller functions (which may call each other). These functions may need to be placed in new modules.

Psects containing data may be reduced when invoking the compiler optimizations, but the effect is less dramatic. The program may need to be rewritten so that it needs less variables. Section 4.4.4 has information on interpreting the map file's call graph if the compiler you are using uses a compiled stack. (If the string `Call graph:` is not present in the map file, then the compiled code uses a hardware stack.) If a data psect needs to be split into smaller psects, the definitions for variables will need to be moved to new modules or more evenly spread in the existing modules. Memory allocation for `auto` variables is entirely handled by the compiler. Other than reducing the number of these variables used, the programmer has little control over their operation. This applies whether the compiled code uses a hardware or compiled stack.

For example, after receiving the message:

```
Can't find 0x34 words (0x34 withtotal) for psect text
 in segment CODE (error)
```

look in the map file for the ranges of unused memory.

```
UNUSED ADDRESS RANGES
        CODE             00000244-0000025F
                         00001000-0000102f
        RAM              00300014-00301FFB
```

In the `CODE` segment, there is 0x1c (0x25f-0x244+1) bytes of space available in one block and 0x30 available in another block. Neither of these are large enough to accommodate the psect `text` which is 0x34 bytes long. Notice, however, that the total amount of memory available is larger than 0x34 bytes.

(492) attempt to position absolute psect "*" is illegal *(Linker)*

This psect is absolute and should not have an address specified in a `-P` option. Either remove the `abs` psect flag, or remove the `-P` linker option.

(493) origin of psect "*" is defined more than once ***(Linker)***

The origin of this psect is defined more than once. There is most likely more than one `-p` linker option specifying this psect.

(494) bad -P format "*/*" ***(Linker)***

The `-P` option given to the linker is malformed. This option specifies placement of a psect, e.g.:

```
-Ptext=10g0h
```

Maybe you meant:

```
-Ptext=10f0h
```

(495) use of both "with=" and "INCLASS/INCLASS" allocation is illegal ***(Linker)***

It is not legal to specify both the link and location of a psect as within a class, when that psect was also defined using a `with` psect flag.

(497) psect "*" exceeds max size: *h > *h ***(Linker)***

The psect has more bytes in it than the maximum allowed as specified using the `size` psect flag.

(498) psect "*" exceeds address limit: *h > *h ***(Linker)***

The maximum address of the psect exceeds the limit placed on it using the `limit` psect flag. Either the psect needs to be linked at a different location or there is too much code/data in the psect.

(499) undefined symbol: ***(Assembler, Linker)***

The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

(500) undefined symbols: ***(Linker)***

A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(501) program entry point is defined more than once ***(Linker)***

There is more than one entry point defined in the object files given the linker. End entry point is specified after the `END` directive. The runtime startup code defines the entry point, e.g.:

```
powerup:
  goto start
  END  powerup  ; end of file and define entry point
; other files that use END should not define another entry point
```

(502) incomplete * record body: length = * ***(Linker)***

An object file contained a record with an illegal size. This probably means the file is truncated or not an object file. Contact HI-TECH Support with details.

(503) ident records do not match ***(Linker)***

The object files passed to the linker do not have matching ident records. This means they are for different processor types.

(504) object code version is greater than *.* ***(Linker)***

The object code version of an object module is higher than the highest version the linker is known to work with. Check that you are using the correct linker. Contact HI-TECH Support if the object file if you have not patched the linker.

(505) no end record found inobject file ***(Linker)***

An object file did not contain an end record. This probably means the file is corrupted or not an object file. Contact HI-TECH Support if the object file was generated by the compiler.

(506) object file record too long: *+* ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(507) unexpected end of file in object file ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(508) relocation offset (*) out of range 0..*-*-1 ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(509) illegal relocation size: * ***(Linker)***

There is an error in the object code format read by the linker. This either means you are using a linker that is out of date, or that there is an internal error in the assembler or linker. Contact HI-TECH Support with details if the object file was created by the compiler.

(510) complex relocation not supported for -R or -L options ***(Linker)***

The linker was given a `-R` or `-L` option with file that contain complex relocation.

(511) bad complex range check ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(512) unknown complex operator 0x* ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(513) bad complex relocation ***(Linker)***

The linker has been asked to perform complex relocation that is not syntactically correct. Probably means an object file is corrupted.

(514) illegal relocation type: * ***(Linker)***

An object file contained a relocation record with an illegal relocation type. This probably means the file is corrupted or not an object file. Contact HI-TECH Support with details if the object file was created by the compiler.

(515) unknown symbol type * ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(516) text record has bad length: *-*-(*+1) < 0 ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(520) function "*" is never called ***(Linker)***

This function is never called. This may not represent a problem, but space could be saved by removing it. If you believe this function should be called, check your source code. Some assembler library routines are never called, although they are actually execute. In this case, the routines are linked in a special sequence so that program execution falls through from one routine to the next.

(521) call depth exceeded by function "*" ***(Linker)***

The call graph shows that functions are nested to a depth greater than specified.

(522) library "*" is badly ordered ***(Linker)***

This library is badly ordered. It will still link correctly, but it will link faster if better ordered.

(523) argument to -W option (*) illegal and ignored ***(Linker)***

The argument to the linker option `-w` is out of range. This option controls two features. For warning levels, the range is -9 to 9. For the map file width, the range is greater than or equal to 10.

(524) unable to open list file "*": * ***(Linker)***

The named list file could not be opened. The linker would be trying to fixup the list file so that it will contain absolute addresses. Ensure that an assembler list file was generated during the compilation stage. Alternatively, remove the assembler list file generation option from the link step.

(525) too many address (memory) spaces; space (*) ignored ***(Linker)***

The limit to the number of address spaces (specified with the `PSECT` assembler directive) is currently 16.

(526) psect "*" not specified in -P option (first appears in "*") ***(Linker)***

This psect was not specified in a `-P` or `-A` option to the linker. It has been linked at the end of the program, which is probably not where you wanted it.

(528) no start record; entry point defaults to zero ***(Linker)***

None of the object files passed to the linker contained a start record. The start address of the program has been set to zero. This may be harmless, but it is recommended that you define a start address in your startup module by using the `END` directive.

(529) usage: objtohex [-Ssymfile] [object-file [hex-file]] ***(Objtohex)***

Improper usage of the command-line tool `objtohex`. If you are invoking `objtohex` directly then please refer to Section 5.11 for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(593) can't find 0x* words (0x* withtotal) for psect "*" in segment "*" ***(Linker)***

See error (491) on Page 427.

(594) undefined symbol: ***(Linker)***

The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

(595) undefined symbols: ***(Linker)***

A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(596) segment "*" (*-*) overlaps segment "*" (*-*) ***(Linker)***

The named segments have overlapping code or data. Check the addresses being assigned by the `-P` linker option.

(599) No psect classes given for COFF write ***(Cromwell)***

Cromwell requires that the program memory psect classes be specified to produce a COFF file. Ensure that you are using the `-N` option as per Section 5.13.2.

(600) No chip arch given for COFF write ***(Cromwell)***

Cromwell requires that the chip architecture be specified to produce a COFF file. Ensure that you are using the -P option as per Section 5.13.1.

(601) Unknown chip arch "*" for COFF write ***(Cromwell)***

The chip architecture specified for producing a COFF file isn't recognised by Cromwell. Ensure that you are using the -P option as per Section 5.13.1 and that the architecture specified matches one of those in Table 5.8.

(602) null file format name ***(Cromwell)***

The `-I` or `-O` option to Cromwell must specify a file format.

(603) ambiguous file format name "*" ***(Cromwell)***

The input or output format specified to Cromwell is ambiguous. These formats are specified with the `-i`*`key`* and `-o`*`key`* options respectively.

(604) unknown file format name "*" ***(Cromwell)***

The output format specified to `CROMWELL` is unknown, e.g.:

```
cromwell -m -P16F877 main.hex main.sym -ocot
```

and output file type of `cot`, did you mean `cof`?

(605) did not recognize format of input file ***(Cromwell)***

The input file to Cromwell is required to be COD, Intel HEX, Motorola HEX, COFF, OMF51, P&E or HI-TECH.

(606) inconsistent symbol tables ***(Cromwell)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(607) inconsistent line number tables ***(Cromwell)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(608) bad path specification ***(Cromwell)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(609) missing processor spec after -P ***(Cromwell)***

The `-p` option to cromwell must specify a processor name.

(610) missing psect classes after -N ***(Cromwell)***

Cromwell requires that the `-N` option be given a list of the names of psect classes.

(611) too many input files ***(Cromwell)***

To many input files have been specified to be converted by `CROMWELL`.

(612) too many output files ***(Cromwell)***

To many output file formats have been specified to `CROMWELL`.

(613) no output file format specified ***(Cromwell)***

The output format must be specified to CROMWELL.

(614) no input files specified ***(Cromwell)***

CROMWELL must have an input file to convert.

(616) option -Cbaseaddr is illegal with options -R or -L ***(Linker)***

The linker option `-Cbaseaddr` cannot be used in conjunction with either the `-R` or `-L` linker options.

(618) error reading COD file data ***(Cromwell)***

An error occurred reading the input COD file. Confirm the spelling and path of the file specified on the command line.

(619) I/O error reading symbol table ***(Cromwell)***

The COD file has an invalid format in the specified record.

(620) filename index out of range in line number record ***(Cromwell)***

The COD file has an invalid value in the specified record.

(621) error writing ELF/DWARF section "*" on "*" ***(Cromwell)***

An error occurred writing the indicated section to the given file. Confirm the spelling and path of the file specified on the command line.

(622) too many type entries ***(Cromwell)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(623) bad class in type hashing *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(624) bad class in type compare *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(625) too many files in COFF file *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(626) string lookup failed in COFF: get_string() *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(627) missing "*" in SDB file "*" line * column * *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(629) bad storage class "*" in SDB file "*" line * column * *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(630) invalid syntax for prefix list in SDB file "*" *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(631) syntax error at token "*" in SDB file "*" line * column * *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(632) can't handle address size (*) *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(633) unknown symbol class (*) *(Cromwell)*

Cromwell has encountered a symbol class in the symbol table of a COFF, Microchip COFF, or ICOFF file which it can't identify.

(634) error dumping "*" *(Cromwell)*

Either the input file to `CROMWELL` is of an unsupported type or that file cannot be dumped to the screen.

(635) invalid HEX file "*" on line * *(Cromwell)*

The specified HEX file contains an invalid line. Contact HI-TECH Support if the HEX file was generated by the compiler.

(636) checksum error in Intel HEX file "*" on line * *(Cromwell, Hexmate)*

A checksum error was found at the specified line in the specified Intel hex file. The HEX file may be corrupt.

(637) unknown prefix "*" in SDB file "*" *(Cromwell)*

This is an internal compiler warning. Contact HI-TECH Software technical support with details.

(638) version mismatch: 0x* expected *(Cromwell)*

The input Microchip COFF file wasn't produced using Cromwell.

(639) zero bit width in Microchip optional header *(Cromwell)*

The optional header in the input Microchip COFF file indicates that the program or data memory spaces are zero bits wide.

(668) prefix list did not match any SDB types *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(669) prefix list matched more than one SDB type *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(670) bad argument to -T *(Clist)*

The argument to the `-T` option to specify tab size was not present or correctly formed. The option expects a decimal interger argument.

(671) argument to -T should be in range 1 to 64 ***(Clist)***

The argument to the `-T` option to specify tab size was not in the expected range. The option expects a decimal interger argument ranging from 1 to 64 inclusive.

(673) missing filename after * option ***(Objtohex)***

The indicated option requires a valid file name. Ensure that the filename argument supplied to this option exists and is spelt correctly.

(674) too many references to "*" ***(Cref)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(679) unknown extraspecial: * ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(680) bad format for -P option ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(682) this architecture is not supported by the PICC Lite compiler ***(Code Generator)***

A target device other than baseline, midrange or highend was specified. This compiler only supports devices from these architecture families.

(683) bank 1 variables are not supported by the PICC Lite compiler ***(Code Generator)***

A variable with an absolute address located in bank 1 was detected. This compiler does not support code generation of variables in this bank.

(684) bank 2 and 3 variables are not supported by the PICC Lite compiler ***(Code Generator)***

A variable with an absolute address located in bank 2 or 3 was detected. This compiler does not support code generation of variables in these banks.

(685) bad putwsize() ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(686) bad switch size (*) ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(687) bad pushreg "*" ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details. See Section 5.7.2 for more information.

(688) bad popreg "*" ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(689) unknown predicate "*" ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(693) interrupt level may only be 0 (default) or 1 ***(Code Generator)***

The only possible interrupt levels are 0 or 1. Check to ensure that all `interrupt_level` pragmas use these levels.

```
#pragma interrupt_level 2  /* oops -- only 0 or 1 */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(695) duplicate case label (*) ***(Code Generator)***

There are two case labels with the same value in this `switch` statement, e.g.:

```
switch(in) {
case '0':  /* if this is case '0'... */
  b++;
  break;
case '0':  /* then what is this case? */
  b--;
  break;
}
```

(696) out-of-range case label (*) *(Code Generator)*

This case label is not a value that the controlling expression can yield, and thus this label will never be selected.

(697) non-constant case label *(Code Generator)*

A `case` label in this `switch` statement has a value which is not a constant.

(698) bit variables must be global or static *(Code Generator)*

A `bit` variable cannot be of type `auto`. If you require a `bit` variable with scope local to a block of code or function, qualify it `static`, e.g.:

```
bit proc(int a)
{
  bit bb;        /* oops --  this should be: static bit bb; */
  bb = (a > 66);
  return bb;
}
```

(699) no case labels in switch *(Code Generator)*

There are no `case` labels in this `switch` statement, e.g.:

```
switch(input) {
}               /* there is nothing to match the value of input */
```

(700) truncation of enumerated value *(Code Generator)*

An enumerated value larger than the maximum value supported by this compiler was detected and has been truncated, e.g.:

```
enum { ZERO, ONE, BIG=0x99999999 } test_case;
```

(701) unreasonable matching depth *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(702) regused(): bad arg to G *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(703) bad GN *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details. See Section 5.7.2 for more information.

(704) bad RET_MASK *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(705) bad which (*) after I *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(706) bad which in expand() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(707) bad SX *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(708) bad mod "+" for how = "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(709) metaregister "*" can't be used directly *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(710) bad U usage *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(711) bad how in expand() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(712) can't generate code for this expression *(Code Generator)*

This error indicates that a C expression is too difficult for the code generator to actually compile. For successful code generation, the code generator must know how to compile an expression and there must be enough resources (e.g. registers or temporary memory locations) available. Simplifying the expression, e.g. using a temporary variable to hold an intermediate result, may get around this message. Contact HI-TECH Support with details of this message.

This error may also be issued if the code being compiled is in some way unusual. For example code which writes to a const-qualified object is illegal and will result in warning messages, but the code generator may unsuccessfully try to produce code to perform the write.

(713) bad initialization list *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(714) bad intermediate code *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(715) bad pragma "*" *(Code Generator)*

The code generator has been passed a `pragma` directive that it does not understand. This implies that the pragma you have used is a HI-TECH specific pragma, but the specific compiler you are using has not implemented this pragma.

(716) bad argument to -M option "*" *(Code Generator)*

The code generator has been passed a `-M` option that it does not understand. This should not happen if it is being invoked by a standard compiler driver.

(718) incompatible intermediate code version; should be *.* *(Code Generator)*

The intermediate code file produced by P1 is not the correct version for use with this code generator. This is either that incompatible versions of one or more compilers have been installed in the same directory, or a temporary file error has occurred leading to corruption of a temporary file. Check the setting of the TEMP environment variable. If it refers to a long path name, change it to something shorter. Contact HI-TECH Support with details if required.

(720) multiple free: * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(721) element count must be constant expression ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(722) bad variable syntax in intermediate code ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(723) function definitions nested too deep ***(Code Generator)***

This error is unlikely to happen with C code, since C cannot have nested functions! Contact HI-TECH Support with details.

(724) bad op (*) in revlog() ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(726) bad op "*" in uconval() ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(727) bad op "*" in bconfloat() ***(Code Generator)***

This is an internal code generator error. Contact HI-TECH technical support with details.

(728) bad op "*" in confloat() ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(729) bad op "*" in conval() ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(730) bad op "*" ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(731) expression error with reserved word ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(732) initialization of bit types is illegal *(Code Generator)*

Variables of type `bit` cannot be initialised, e.g.:

```
bit b1 = 1; /* oops!
               b1 must be assigned after its definition */
```

(733) bad string "*" in pragma "psect" *(Code Generator)*

The code generator has been passed a `pragma psect` directive that has a badly formed string, e.g.:

```
#pragma psect text  /* redirect text psect into what? */
```

Maybe you meant something like:

```
#pragma psect text=special_text
```

(734) too many "psect" pragmas *(Code Generator)*

Too many `#pragma psect` directives have been used.

(735) bad string "*" in pragma "stack_size" *(Code Generator)*

The argument to the stack_size pragma is malformed. This pragma must be followed by a number representing the maximum allowed stack size.

(737) unknown argument "*" to pragma "switch" *(Code Generator)*

The `#pragma switch` directive has been used with an invalid switch code generation method. Possible arguments are: `auto`, `simple` and `direct`.

(739) error closing output file *(Code Generator, Optimiser)*

The compiler detected an error when closing a file. Contact HI-TECH Support with details.

(740) zero dimension array is illegal *(Code Generator)*

The code generator has been passed a declaration that results in an array having a zero dimension.

(741) bitfield too large (* bits) ***(Code Generator)***

The maximum number of bits in a bit field is the same as the number of bits in an `int`, e.g. assuming an `int` is 16 bits wide:

```
struct {
  unsigned flag  : 1;
  unsigned value : 12;
  unsigned cont  : 6;   /* oops -- that's a total of 19 bits */
} object;
```

(742) function "*" argument evaluation overlapped ***(Linker)***

A function call involves arguments which overlap between two functions. This could occur with a call like:

```
void fn1(void)
{
  fn3( 7, fn2(3), fn2(9));    /* Offending call */
}
char fn2(char fred)
{
  return fred + fn3(5,1,0);
}
char fn3(char one, char two, char three)
{
  return one+two+three;
}
```

where `fn1` is calling `fn3`, and two arguments are evaluated by calling `fn2`, which in turn calls `fn3`. The program structure should be modified to prevent this type of call sequence.

(743) divide by zero ***(Code Generator)***

An expression involving a division by zero has been detected in your code.

(744) static object "*" has zero size ***(Code Generator)***

A static object has been declared, but has a size of zero.

(745) nodecount = * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(746) object "*" qualified const, but not initialized *(Code Generator)*

An object has been qualified as `const`, but there is no initial value supplied at the definition. As this object cannot be written by the C program, this may imply the intial value was accidently omitted.

(747) unrecognized option "*" to -Z *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(748) variable "*" may be used before set *(Code Generator)*

This variable may be used before it has been assigned a value. Since it is an `auto` variable, this will result in it having a random value, e.g.:

```
void main(void)
{
  int a;
  if(a)         /* oops -- a has never been assigned a value */
    process();
}
```

(749) unknown register name "*" used with pragma *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(750) constant operand to || or && *(Code Generator)*

One operand to the logical operators `||` or `&&` is a constant. Check the expression for missing or badly placed parentheses. This message may also occur if the global optimizer is enabled and one of the operands is an `auto` or `static` local variable whose value has been tracked by the code generator, e.g.:

```
{
int a;
a = 6;
if(a || b)  /* a is 6, therefore this is always true */
    b++;
```

(751) arithmetic overflow in constant expression *(Code Generator)*

A constant expression has been evaluated by the code generator that has resulted in a value that is too big for the type of the expression. The most common code to trigger this warning is assignments to `signed` data types. For example:

```
signed char c;
c = 0xFF;
```

As a `signed` 8-bit quantity, `c` can only be assigned values -128 to 127. The constant is equal to 255 and is outside this range. If you mean to set all bits in this variable, then use either of:

```
c = ~0x0;
c = -1;
```

which will set all the bits in the variable regardless of the size of the variable and without warning.

This warning can also be triggered by intermediate values overflowing. For example:

```
unsigned int i;  /* assume ints are 16 bits wide */
i = 240 * 137;   /* this should be okay, right? */
```

A quick check with your calculator reveals that 240 * 137 is 32880 which can easily be stored in an `unsigned int`, but a warning is produced. Why? Because 240 and 137 and both `signed int` values. Therefore the result of the multiplication must also be a `signed int` value, but a `signed int` cannot hold the value 32880. (Both operands are constant values so the code generator can evaluate this expression at compile time, but it must do so following all the ANSI rules.) The following code forces the multiplication to be performed with an `unsigned` result:

```
i = 240u * 137;  /* force at least one operand
                    to be unsigned */
```

(752) conversion to shorter data type *(Code Generator)*

Truncation may occur in this expression as the lvalue is of shorter type than the rvalue, e.g.:

```
char a;
int b, c;
a = b + c;  /* int to char conversion
               may result in truncation */
```

(753) undefined shift (* bits) ***(Code Generator)***

An attempt has been made to shift a value by a number of bits equal to or greater than the number of bits in the data type. This will produce an undefined result on many processors. This is non-portable code and is flagged as having undefined results by the C Standard, e.g.:

```
int input;
input <<= 33;  /* oops -- that shifts the entire value out */
```

(754) bitfield comparison out of range ***(Code Generator)***

This is the result of comparing a bitfield with a value when the value is out of range of the bitfield. For example, comparing a 2-bit bitfield to the value 5 will never be true as a 2-bit bitfield has a range from 0 to 3, e.g.:

```
struct {
  unsigned mask : 2;  /* mask can hold values 0 to 3 */
} value;
int compare(void)
{
  return (value.mask == 6);  /* test can
}
```

(755) divide by zero ***(Code Generator)***

A constant expression that was being evaluated involved a division by zero, e.g.:

```
a /= 0;  /* divide by 0: was this what you were intending */
```

(757) constant conditional branch ***(Code Generator)***

A conditional branch (generated by an `if`, `for`, `while` statement etc.) always follows the same path. This will be some sort of comparison involving a variable and a constant expression. For the code generator to issue this message, the variable must have local scope (either `auto` or `static` local) and the global optimizer must be enabled, possibly at higher level than 1, and the warning level threshold may need to be lower than the default level of 0.

The global optimizer keeps track of the contents of local variables for as long as is possible during a function. For C code that compares these variables to constants, the result of the comparison can be deduced at compile time and the output code hard coded to avoid the comparison, e.g.:

```
{
  int a, b;
  a = 5;
  /* this can never be false;
     always perform the true statement */
  if(a == 4)
    b = 6;
```

will produce code that sets `a` to 5, then immediately sets `b` to 6. No code will be produced for the comparison `if(a == 4)`. If a was a global variable, it may be that other functions (particularly interrupt functions) may modify it and so tracking the variable cannot be performed.

This warning may indicate more than an optimization made by the compiler. It may indicate an expression with missing or badly placed parentheses, causing the evaluation to yield a value different to what you expected.

This warning may also be issued because you have written something like `while(1)`. To produce an infinite loop, use `for(;;)`.

A similar situation arises with for loops, e.g.:

```
{
  int a, b;
  /* this loop must iterate at least once */
  for(a=0; a!=10; a++)
    b = func(a);
```

In this case the code generator can again pick up that `a` is assigned the value 0, then immediately checked to see if it is equal to 10. Because `a` is modified during the `for` loop, the comparison code cannot be removed, but the code generator will adjust the code so that the comparison is not performed on the first pass of the loop; only on the subsequent passes. This may not reduce code size, but it will speed program execution.

(758) constant conditional branch: possible use of "=" instead of "==" *(Code Generator)*

There is an expression inside an `if` or other conditional construct, where a constant is being assigned to a variable. This may mean you have inadvertently used an assignment = instead of a compare ==, e.g.:

```
int a, b;
/* this can never be false;
   always perform the true statement */
if(a = 4)
  b = 6;
```

will assign the value 4 to a, then , as the value of the assignment is always true, the comparison can be omitted and the assignment to `b` always made. Did you mean:

```
/* this can never be false;
   always perform the true statement */
if(a == 4)
    b = 6;
```

which checks to see if a is equal to 4.

(759) expression generates no code *(Code Generator)*

This expression generates no output code. Check for things like leaving off the parentheses in a function call, e.g.:

```
int fred;
fred;     /* this is valid, but has no effect at all */
```

Some devices require that special function register need to be read to clear hardware flags. To accommodate this, in some instances the code generator *does* produce code for a statement which only consists of a variable ID. This may happen for variables which are qualified as `volatile`. Typically the output code will read the variable, but not do anything with the value read.

(760) portion of expression has no effect *(Code Generator)*

Part of this expression has no side effects, and no effect on the value of the expression, e.g.:

```
int a, b, c;
a = b,c;  /* "b" has no effect,
             was that meant to be a comma? */
```

(761) sizeof yields 0 *(Code Generator)*

The code generator has taken the size of an object and found it to be zero. This almost certainly indicates an error in your declaration of a pointer, e.g. you may have declared a pointer to a zero length array. In general, pointers to arrays are of little use. If you require a pointer to an array of objects of unknown length, you only need a pointer to a single object that can then be indexed or incremented.

(762) constant truncated when assigned to bitfield *(Code Generator)*

A constant value is too large for a bitfield structure member to which it is being assigned, e.g.

```
struct INPUT {
  unsigned a : 3;
  unsigned b : 5;
} input_grp;
input_grp.a = 0x12;
/* 12h cannot fit into a 3-bit wide object */
```

(763) constant left operand to "? :" operator *(Code Generator)*

The left operand to a conditional operator `?` is constant, thus the result of the tertiary operator `?:` will always be the same, e.g.:

```
a = 8 ? b : c;  /* this is the same as saying a = b; */
```

(764) mismatched comparison *(Code Generator)*

A comparison is being made between a variable or expression and a constant value which is not in the range of possible values for that expression, e.g.:

```
unsigned char c;
if(c > 300)      /* oops -- how can this be true? */
  close();
```

(765) degenerate unsigned comparison *(Code Generator)*

There is a comparison of an `unsigned` value with zero, which will always be true or false, e.g.:

```
unsigned char c;
if(c >= 0)
```

will always be true, because an `unsigned` value can never be less than zero.

(766) degenerate signed comparison *(Code Generator)*

There is a comparison of a `signed` value with the most negative value possible for this type, such that the comparison will always be true or false, e.g.:

```
char c;
if(c >= -128)
```

will always be true, because an 8 bit signed char has a maximum negative value of -128.

(767) constant truncated to bitfield width ***(Code Generator)***

A constant value is too large for a bitfield structure member on which it is operating, e.g.

```
struct INPUT {
  unsigned a : 3;
  unsigned b : 5;
} input_grp;
input_grp.a |= 0x13;
/* 13h to large for 3-bit wide object */
```

(768) constant relational expression ***(Code Generator)***

There is a relational expression that will always be true or false. This may be because e.g. you are comparing an `unsigned` number with a negative value, or comparing a variable with a value greater than the largest number it can represent, e.g.:

```
unsigned int a;
if(a == -10)    /* if a is unsigned, how can it be -10? */
  b = 9;
```

(769) no space for macro definition ***(Assembler)***

The assembler has run out of memory.

(772) include files nested too deep ***(Assembler)***

Macro expansions and include file handling have filled up the assembler's internal stack. The maximum number of open macros and include files is 30.

(773) macro expansions nested too deep ***(Assembler)***

Macro expansions in the assembler are nested too deep. The limit is 30 macros and include files nested at one time.

(774) too many macro parameters *(Assembler)*

There are too many macro parameters on this macro definition.

(776) can't allocate space for object "*" (offs: *) *(Assembler)*

The assembler has run out of memory.

(777) can't allocate space for opnd structure within object "*", (offs: *) *(Assembler)*

The assembler has run out of memory.

(780) too many psects defined *(Assembler)*

There are too many psects defined! Boy, what a program!

(781) can't enter abs psect *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(782) REMSYM error *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(783) "with" psects are cyclic *(Assembler)*

If Psect A is to be placed "with" Psect B, and Psect B is to be placed "with" Psect A, there is no hierarchy. The `with` flag is an attribute of a psect and indicates that this psect must be placed in the same memory page as the specified psect.

Remove a `with` flag from one of the psect declarations. Such an assembler declaration may look like:

```
psect my_text,local,class=CODE,with=basecode
```

which will define a psect called `my_text` and place this in the same page as the psect `basecode`.

(784) overfreed *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(785) too many temporary labels *(Assembler)*

There are too many temporary labels in this assembler file. The assembler allows a maximum of 2000 temporary labels.

(787) can't handle "v_rtype" of * in copyexpr *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(788) invalid character "*" in number *(Assembler)*

A number contained a character that was not part of the range 0-9 or 0-F.

(790) end of file inside conditional *(Assembler)*

END-of-FILE was encountered while scanning for an "endif" to match a previous "if".

(793) unterminated macro argument *(Assembler)*

An argument to a macro is not terminated. Note that angle brackets ("< >") are used to quote macro arguments.

(794) invalid number syntax *(Assembler, Optimiser)*

The syntax of a number is invalid. This can be, e.g. use of 8 or 9 in an octal number, or other malformed numbers.

(796) use of LOCAL outside macros is illegal *(Assembler)*

The `LOCAL` directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.

(797) syntax error in LOCAL argument *(Assembler)*

A symbol defined using the `LOCAL` assembler directive in an assembler macro is syntactically incorrect. Ensure that all symbols and all other assembler identifiers conform with the assembly language of the target device.

(798) macro argument may not appear after LOCAL *(Assembler)*

The list of labels after the directive LOCAL may not include any of the formal parameters to the macro, e.g.:

```
mmm macro a1
  move  r0, #a1
  LOCAL a1  ; oops --
            ; the macro parameter cannot be used with local
ENDM
```

(799) REPT argument must be >= 0 *(Assembler)*

The argument to a REPT directive must be greater than zero, e.g.:

```
rept -2            ; -2 copies of this code? */
  move r0, [r1]++
endm
```

(800) undefined symbol "*" *(Assembler)*

The named symbol is not defined in this module, and has not been specified GLOBAL.

(801) range check too complex *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(802) invalid address after END directive *(Assembler)*

The start address of the program which is specified after the assembler END directive must be a label in the current file.

(803) undefined temporary label *(Assembler)*

A temporary label has been referenced that is not defined. Note that a temporary label must have a number >= 0.

(804) write error on object file *(Assembler)*

The assembler failed to write to an object file. This may be an internal compiler error. Contact HI-TECH Software technical support with details.

(805) non-whitespace ignored after END directive *(Assembler)*

The `END` directive, if used, indicates the end of the source file. If there are non-whitespace characters after the `END` directive, then the directive is does actually mark the end of the file.

(806) attempted to get an undefined object (*) *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(807) attempted to set an undefined object (*) *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(808) bad size in add_reloc() *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(809) unknown addressing mode (*) *(Assembler, Optimiser)*

An unknown addressing mode was used in the assembly file.

(811) "cnt" too large (*) in display() *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(814) processor type not defined *(Assembler)*

The processor must be defined either from the command line (eg. -16c84), via the PROCESSOR assembler directive, or via the LIST assembler directive.

(815) syntax error in chipinfo file at line * *(Assembler)*

The chipinfo file contains non-standard syntax at the specified line.

(816) duplicate ARCH specification in chipinfo file "*" at line * *(Assembler, Driver)*

The chipinfo file has a processor section with multiple ARCH values. Only one ARCH value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(817) unknown architecture in chipinfo file at line * *(Assembler, Driver)*

An chip architecture (family) that is unknown was encountered when reading the chip INI file.

(818) duplicate BANKS for "*" in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple BANKS values. Only one BANKS value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(819) duplicate ZEROREG for "*" in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple ZEROREG values. Only one ZEROREG value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(820) duplicate SPAREBIT for "*" in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple SPAREBIT values. Only one SPAREBIT value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(822) duplicate ROMSIZE for "*" in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple ROMSIZE values. Only one ROMSIZE value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(823) duplicate START for "*" in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple START values. Only one START value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(824) duplicate LIB for "*" in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple `LIB` values. Only one `LIB` value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(825) too many RAMBANK lines in chipinfo file for "*" ***(Assembler)***

The chipinfo file contains a processor section with too many RAMBANK fields. Reduce the number of values.

(826) inverted ram bank in chipinfo file at line * ***(Assembler, Driver)***

The second hex number specified in the RAM field in the chipinfo file must be greater in value than the first.

(827) too many COMMON lines in chipinfo file for "*" *(Assembler)*

There are too many lines specifying common (access bank) memory in the chip configuration file.

(828) inverted common bank in chipinfo file at line * *(Assembler, Driver)*

The second hex number specified in the COMMON field in the chipinfo file must be greater in value than the first. Contact HI-TECH Support if you have not modified the chipinfo INI file.

(829) unrecognized line in chipinfo file at line * *(Assembler)*

The chipinfo file contains a processor section with an unrecognised line. Contact HI-TECH Support if the INI has not been edited.

(830) missing ARCH specification for "*" in chipinfo file *(Assembler)*

The chipinfo file has a processor section without an ARCH values. The architecture of the processor must be specified. Contact HI-TECH Support if the chipinfo file has not been modified.

(832) empty chip info file "*" *(Assembler)*

The chipinfo file contains no data. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(833) no valid entries in chipinfo file *(Assembler)*

The chipinfo file contains no valid processor descriptions.

(834) page width must be >= 60 *(Assembler)*

The listing page width must be at least 60 characters. Any less will not allow a properly formatted listing to be produced, e.g.:

```
LIST C=10  ; the page width will need to be wider than this
```

(835) form length must be >= 15 *(Assembler)*

The form length specified using the `-Flength` option must be at least 15 lines. Setting this length to zero is allowed and turns off paging altogether. The default value is zero (pageless).

(836) no file arguments *(Assembler)*

The assembler has been invoked without any file arguments. It cannot assemble anything.

(839) relocation too complex *(Assembler)*

The complex relocation in this expression is too big to be inserted into the object file.

(840) phase error *(Assembler)*

The assembler has calculated a different value for a symbol on two different passes. This is probably due to bizarre use of macros or conditional assembly.

(842) bad bit number *(Assembler, Optimiser)*

A bit number must be an absolute expression in the range 0-7.

(843) a macro name can't also be an EQU/SET symbol *(Assembler)*

An EQU or SET symbol has been found with the same name as a macro. This is not allowed. For example:

```
getval MACRO
  mov  r0, r1
ENDM
getval EQU 55h  ; oops -- choose a different name to the macro
```

(844) lexical error *(Assembler, Optimiser)*

An unrecognized character or token has been seen in the input.

(845) symbol "*" defined more than once *(Assembler)*

This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, e.g.:

```
_next:
  move r0, #55
  move [r1], r0
_next:           ; oops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an *underscore* prepended to their name after compilation.

(846) relocation error ***(Assembler, Optimiser)***

It is not possible to add together two relocatable quantities. A constant may be added to a relocatable value, and two relocatable addresses in the same psect may be subtracted. An absolute value must be used in various places where the assembler must know a value at assembly time.

(847) operand error ***(Assembler, Optimiser)***

The operand to this opcode is invalid. Check your assembler reference manual for the proper form of operands for this instruction.

(849) illegal instruction for this processor ***(Assembler)***

The instruction is not supported by this processor.

(850) PAGESEL not usable with this processor ***(Assembler)***

The PAGESEL pseudo-instruction is not usable with the device selected.

(852) radix must be from 2 - 16 ***(Assembler)***

The radix specified using the `RADIX` assembler directive must be in the range from 2 (binary) to 16 (hexadecimal).

(853) invalid size for FNSIZE directive ***(Assembler)***

The assembler FNSIZE assembler directive arguments must be positive constants.

(855) ORG argument must be a positive constant ***(Assembler)***

An argument to the `ORG` assembler directive must be a positive constant or a symbol which has been equated to a positive constant, e.g.:

```
ORG -10  /* this must a positive offset to the current psect */
```

(856) ALIGN argument must be a positive constant *(Assembler)*

The `align` assembler directive requires a non-zero positive integer argument.

(857) psect may not be local and global *(Linker)*

A local psect may not have the same name as a global psect, e.g.:

```
psect text,class=CODE       ; text is implicitly global
  move r0, r1
; elsewhere:
psect text,local,class=CODE
  move r2, r4
```

The `global` flag is the default for a psect if its scope is not explicitly stated.

(859) argument to C option must specify a positive constant *(Assembler)*

The parameter to the `LIST` assembler control's `C=` option (which sets the column width of the listing output) must be a positive decimal constant number, e.g.:

```
LIST C=a0h  ; constant must be decimal and positive,
              try: LIST C=80
```

(860) page width must be >= 49 *(Assembler)*

The page width suboption to the `LIST` assembler directive must specify a with of at least 49.

(861) argument to N option must specify a positive constant *(Assembler)*

The parameter to the `LIST` assembler control's `N` option (which sets the page length for the listing output) must be a positive constant number, e.g.:

```
LIST N=-3  ; page length must be positive
```

(862) symbol is not external *(Assembler)*

A symbol has been declared as EXTRN but is also defined in the current module.

(863) symbol can't be both extern and public *(Assembler)*

If the symbol is declared as extern, it is to be imported. If it is declared as public, it is to be exported from the current module. It is not possible for a symbol to be both.

(864) argument to "size" psect flag must specify a positive constant ***(Assembler)***

The parameter to the `PSECT` assembler directive's `size` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,size=-200  ; a negative size?
```

(865) psect flag "size" redefined ***(Assembler)***

The `size` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,size=400
; elsewhere:
psect spdata,class=RAM,size=500
```

(866) argument to "reloc" psect flag must specify a positive constant ***(Assembler)***

The parameter to the `PSECT` assembler directive's `reloc` option must be a positive constant number, e.g.:

```
psect test,class=CODE,reloc=-4  ; the reloc must be positive
```

(867) psect flag "reloc" redefined ***(Assembler)***

The `reloc` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,reloc=4
; elsewhere:
psect spdata,class=RAM,reloc=8
```

(868) argument to "delta" psect flag must specify a positive constant ***(Assembler)***

The parameter to the `PSECT` assembler directive's `DELTA` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,delta=-2  ; negative delta value doesn't make sense
```

(869) psect flag "delta" redefined ***(Assembler)***

The 'DELTA' option of a psect has been redefined more than once in the same module.

(870) argument to "pad" psect flag must specify a positive constant ***(Assembler)***

The parameter to the PSECT assembler directive's 'PAD' option must be a non-zero positive integer.

(871) argument to "space" psect flag must specify a positive constant ***(Assembler)***

The parameter to the PSECT assembler directive's `space` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,space=-1  ; space values start at zero
```

(872) psect flag "space" redefined ***(Assembler)***

The `space` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,space=0
; elsewhere:
psect spdata,class=RAM,space=1
```

(873) a psect may only be in one class ***(Assembler)***

You cannot assign a psect to more than one class. The psect was defined differently at this point than when it was defined elsewhere. A psect's class is specified via a flag as in the following:

```
psect text,class=CODE
```

Look for other psect definitions that specify a different class name.

(874) a psect may only have one "with" option ***(Assembler)***

A psect can only be placed `with` one other psect. A psect's with option is specified via a flag as in the following:

```
psect bss,with=data
```

Look for other psect definitions that specify a different `with` psect name.

(875) bad character constant in expression ***(Assembler,Optimizer)***

The character constant was expected to consist of only one character, but was found to be greater than one character or none at all. An assembler specific example:

```
mov  r0, #'12'    ; '12' specifies two characters
```

(876) syntax error *(Assembler, Optimiser)*

A syntax error has been detected. This could be caused a number of things.

(877) yacc stack overflow *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(878) -S option used: "*" ignored *(Driver)*

The indicated assembly file has been supplied to the driver in conjunction with the `-S` option. The driver really has nothing to do since the file is already an assembly file.

(880) invalid number of parameters. Use "* –HELP" for help *(Driver)*

Improper command-line usage of the of the compiler's driver.

(881) setup succeeded *(Driver)*

The compiler has been successfully setup using the `--setup` driver option.

(883) setup failed *(Driver)*

The compiler was not successfully setup using the `--setup` driver option. Ensure that the directory argument to this option is spelt correctly, is syntactically correct for your host operating system and it exists.

(884) please ensure you have write permissions to the configuration file *(Driver)*

The compiler was not successfully setup using the `--setup` driver option because the driver was unable to access the XML configuration file. Ensure that you have write permission to this file. The driver will search the following configuration files in order:

- the file specified by the environment variable `HTC_XML`
- the file `/etc/htsoft.xml` if the directory '`/etc`' is writable and there is no `.htsoft.xml` file in your home directory
- the file `.htsoft.xml` file in your home directory

If none of the files can be located then the above error will occur.

(889) this * compiler has expired ***(Driver)***

The demo period for this compiler has concluded.

(890) contact HI-TECH Software to purchase and re-activate this compiler ***(Driver)***

The evaluation period of this demo installation of the compiler has expired. You will need to purchase the compiler to re-activate it. If however you sincerely believe the evaluation period has ended prematurely please contact HI-TECH technical support.

(891) can't open psect usage map file "*": * ***(Driver)***

The driver was unable to open the indicated file. The psect usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(892) can't open memory usage map file "*": * ***(Driver)***

The driver was unable to open the indicated file. The memory usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(893) can't open HEX usage map file "*": * ***(Driver)***

The driver was unable to open the indicated file. The HEX usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(894) unknown source file type "*" ***(Driver)***

The extension of the indicated input file could not be determined. Only files with the extensions `as`, `c`, `obj`, `usb`, `p1`, `lib` or `hex` are identified by the driver.

(895) can't request and specify options in the one command ***(Driver)***

The usage of the driver options `--getoption` and `--setoption` is mutually exclusive.

(896) no memory ranges specified for data space ***(Driver)***

No on-chip or external memory ranges have been specified for the data space memory for the device specified.

(897) no memory ranges specified for program space ***(Driver)***

No on-chip or external memory ranges have been specified for the program space memory for the device specified.

(899) can't open option file "*" for application "*": * ***(Driver)***

An option file specified by a `--getoption` or `--setoption` driver option could not be opened. If you are using the `--setoption` option ensure that the name of the file is spelt correctly and that it exists. If you are using the `--getoption` option ensure that this file can be created at the given location or that it is not in use by any other application.

(900) exec failed: * ***(Driver)***

The subcomponent listed failed to execute. Does the file exist? Try re-installing the compiler.

(902) no chip name specified; use "* –CHIPINFO" to see available chip names ***(Driver)***

The driver was invoked without selecting what chip to build for. Running the driver with the –CHIPINFO option will display a list of all chips that could be selected to build for.

(904) illegal format specified in "*" option ***(Driver)***

The usage of this option was incorrect. Confirm correct usage with –HELP or refer to the part of the manual that discusses this option.

(905) illegal application specified in "*" option ***(Driver)***

The application given to this option is not understood or does not belong to the compiler.

(907) unknown memory space tag "*" in "*" option specification ***(Driver)***

A parameter to this memory option was a string but did not match any valid *tags*. Refer to the section of this manual that describes this option to see what tags (if any) are valid for this device.

(908) exit status = * ***(Driver)***

One of the subcomponents being executed encountered a problem and returned an error code. Other messages should have been reported by the subcomponent to explain the problem that was encountered.

(913) "*" option may cause compiler errors in some standard header files ***(Driver)***

Using this option will invalidate some of the qualifiers used in the standard header files resulting in errors. This issue and its solution are detailed in the section of this manual that specifically discusses this option.

(915) no room for arguments ***(Preprocessor, Parser, Code Generator, Linker, Objtohex)***

The code generator could not allocate any more memory.

(917) argument too long ***(Preprocessor, Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(918) *: no match ***(Preprocessor, Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(919) * in chipinfo file "*" at line * ***(Driver)***

The specified parameter in the chip configuration file is illegal.

(920) empty chipinfo file ***(Driver, Assembler)***

The chip configuration file was able to be opened but it was empty. Try re-installing the compiler.

(922) chip "*" not present in chipinfo file "*" ***(Driver)***

The chip selected does not appear in the compiler's chip configuration file. You may need to contact HI-TECH Software to see if support for this device is available or upgrade the version of your compiler.

(923) unknown suboption "*" ***(Driver)***

This option can take suboptions, but this suboption is not understood. This may just be a simple spelling error. If not, –HELP to look up what suboptions are permitted here.

(924) missing argument to "*" option ***(Driver)***

This option expects more data but none was given. Check the usage of this option.

(925) extraneous argument to "*" option ***(Driver)***

This option does not accept additional data, yet additional data was given. Check the usage of this option.

(926) duplicate "*" option ***(Driver)***

This option can only appear once, but appeared more than once.

(928) bad "*" option value ***(Driver, Assembler)***

The indicated option was expecting a valid hexadecimal integer argument.

(929) bad "*" option ranges ***(Driver)***

This option was expecting a parameter in a range format (*start_of_range-end_of_range*), but the parameter did not conform to this syntax.

(930) bad "*" option specification ***(Driver)***

The parameters to this option were not specified correctly. Run the driver with –HELP or refer to the driver's chapter in this manual to verify the correct usage of this option.

(931) command file not specified ***(Driver)***

Command file to this application, expected to be found after `'@'` or `'<'` on the command line was not found.

(939) no file arguments ***(Driver)***

The driver has been invoked with no input files listed on its command line. If you are getting this message while building through a third party IDE, perhaps the IDE could not verify the source files to compile or object files to link and withheld them from the command line.

(940) *-bit checksum * placed at * ***(Objtohex)***

Presenting the result of the requested checksum calculation.

(941) bad "*" assignment; USAGE: ** ***(Hexmate)***

An option to Hexmate was incorrectly used or incomplete. Follow the usage supplied by the message and ensure that that the option has been formed correctly and completely.

(942) unexpected character on line * of file "*" ***(Hexmate)***

File contains a character that was not valid for this type of file, the file may be corrupt. For example, an Intel hex file is expected to contain only ASCII representations of hexadecimal digits, colons (:) and line formatting. The presence of any other characters will result in this error.

(944) data conflict at address *h between * and * ***(Hexmate)***

Sources to Hexmate request differing data to be stored to the same address. To force one data source to override the other, use the '+' specifier. If the two named sources of conflict are the same source, then the source may contain an error.

(945) checksum range (*h to *h) contained an indeterminate value ***(Hexmate)***

The range for this checksum calculation contained a value that could not be resolved. This can happen if the checksum result was to be stored within the address range of the checksum calculation.

(948) checksum result width must be between 1 and 4 bytes ***(Hexmate)***

The requested checksum byte size is illegal. Checksum results must be within 1 to 4 bytes wide. Check the parameters to the -CKSUM option.

(949) start of checksum range must be less than end of range ***(Hexmate)***

The -CKSUM option has been given a range where the start is greater than the end. The parameters may be incomplete or entered in the wrong order.

(951) start of fill range must be less than end of range ***(Hexmate)***

The -FILL option has been given a range where the start is greater than the end. The parameters may be incomplete or entered in the wrong order.

(953) unknown -HELP sub-option: * ***(Hexmate)***

Invalid sub-option passed to -HELP. Check the spelling of the sub-option or use -HELP with no sub-option to list all options.

(956) -SERIAL value must be between 1 and * bytes long ***(Hexmate)***

The serial number being stored was out of range. Ensure that the serial number can be stored in the number of bytes permissible by this option.

(958) too many input files specified; * file maximum ***(Hexmate)***

Too many file arguments have been used. Try merging these files in several stages rather than in one command.

(960) unexpected record type (*) on line * of "*" ***(Hexmate)***

Intel hex file contained an invalid record type. Consult the Intel hex format specification for valid record types.

(962) forced data conflict at address *h between * and * ***(Hexmate)***

Sources to Hexmate force differing data to be stored to the same address. More than one source using the '+' specifier store data at the same address. The actual data stored there may not be what you expect.

(963) checksum range includes voids or unspecified memory locations ***(Hexmate)***

Checksum range had gaps in data content. The runtime calculated checksum is likely to differ from the compile-time checksum due to gaps/unused byes within the address range that the checksum is calculated over. Filling unused locations with a known value will correct this.

(964) unpaired nibble in -FILL value will be truncated ***(Hexmate)***

The hexadecimal code given to the FILL option contained an incomplete byte. The incomplete byte (nibble) will be disregarded.

(965) -STRPACK option not yet implemented, option will be ignored ***(Hexmate)***

This option currently is not available and will be ignored.

(966) no END record for HEX file "*" ***(Hexmate)***

Intel hex file did not contain a record of type END. The hex file may be incomplete.

(967) unused function definition "*" (from line *) ***(Parser)***

The indicated `static` function was never called in the module being compiled. Being static, the function cannot be called from other modules so this warning implies the function is never used. Either the function is redundant, or the code that was meant to call it was excluded from compilation or misspelt the name of the function.

(968) unterminated string ***(Assembler, Optimiser)***

A string constant appears not to have a closing quote missing.

(969) end of string in format specifier ***(Parser)***

The format specifier for the printf() style function is malformed.

(970) character not valid at this point in format specifier ***(Parser)***

The printf() style format specifier has an illegal character.

(971) type modifiers not valid with this format ***(Parser)***

Type modifiers may not be used with this format.

(972) only modifiers "h" and "l" valid with this format ***(Parser)***

Only modifiers `h` (`short`) and `l` (`long`) are legal with this `printf` format specifier.

(973) only modifier "l" valid with this format ***(Parser)***

The only modifier that is legal with this format is `l` (for long).

(974) type modifier already specified ***(Parser)***

This type modifier has already be specified in this type.

(975) invalid format specifier or type modifier ***(Parser)***

The format specifier or modifier in the printf-style string is illegal for this particular format.

(976) field width not valid at this point ***(Parser)***

A field width may not appear at this point in a printf() type format specifier.

(978) this identifier is already an enum tag ***(Parser)***

This identifier following a `struct` or `union` keyword is already the tag for an enumerated type, and thus should only follow the keyword `enum`, e.g.:

```
enum IN {ONE=1, TWO};
struct IN {             /* oops -- IN is already defined */
  int a, b;
};
```

(979) this identifier is already a struct tag ***(Parser)***

This identifier following a `union` or `enum` keyword is already the tag for a structure, and thus should only follow the keyword `struct`, e.g.:

```
struct IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(980) this identifier is already a union tag ***(Parser)***

This identifier following a `struct` or `enum` keyword is already the tag for a `union`, and thus should only follow the keyword `union`, e.g.:

```
union IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(981) pointer required ***(Parser)***

A pointer is required here, e.g.:

```
struct DATA data;
data->a = 9;      /* data is a structure,
                     not a pointer to a structure */
```

(982) unknown op "*" in nxtuse() ***(Optimiser,Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(983) storage class redeclared ***(Parser)***

A variable previously declared as being *static*, has now be redeclared as *extern*.

(984) type redeclared ***(Parser)***

The type of this function or object has been redeclared. This can occur because of two incompatible declarations, or because an implicit declaration is followed by an incompatible declaration, e.g.:

```
int a;
char a;  /* oops -- what is the correct type? */
```

(985) qualifiers redeclared ***(Parser)***

This function or variable has different qualifiers in different declarations.

(986) enum member redeclared ***(Parser)***

A member of an enumeration is defined twice or more with differing values. Does the member appear twice in the same list or does the name of the member appear in more than one enum list?

(987) arguments redeclared ***(Parser)***

The data types of the parameters passed to this function do not match its prototype.

(988) number of arguments redeclared ***(Parser)***

The number of arguments in this function declaration does not agree with a previous declaration of the same function.

(989) module has code below file base of *h ***(Linker)***

This module has code below the address given, but the `-C` option has been used to specify that a binary output file is to be created that is mapped to this address. This would mean code from this module would have to be placed before the beginning of the file! Check for missing psect directives in assembler files.

(990) modulus by zero in #if; zero result assumed ***(Preprocessor)***

A modulus operation in a `#if` expression has a zero divisor. The result has been assumed to be zero, e.g.:

```
#define ZERO 0
#if FOO%ZERO    /* this will have an assumed result of 0 */
  #define INTERESTING
#endif
```

(991) integer expression required ***(Parser)***

In an `enum` declaration, values may be assigned to the members, but the expression must evaluate to a constant of type `int`, e.g.:

```
enum {one = 1, two, about_three = 3.12};
  /* no non-int values allowed */
```

(992) can't find op ***(Assembler, Optimiser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(993) some command-line options are disabled ***(Driver)***

The compiler is operating in demo mode. Some command-line options are disabled.

(994) some command-line options are disabled and compilation is delayed ***(Driver)***

The compiler is operating in demo mode. Some command-line options are disabled, the compilation speed will be slower.

(995) some command-line options are disabled, code size is limited to 16kB, compilation is delayed ***(Driver)***

The compiler is operating in demo mode. Some command-line options are disabled, the compilation speed will be slower, and the maximum allowed code size is limited to 16kB.

(1003) nested "if" directives too deep ***(Assembler)***

A series of assembler IF directives have been nested too deep. The maximum depth may vary but typically 10 levels are permitted.

(1013) argument to "limit" psect flag must specify a positive constant ***(Assembler)***

The value of the *limit* flag as used in a psect's declaration must be a positive constant. A negative *limit* is not permissible.

(1014) psect flag "limit" redefined ***(Assembler)***

The limit flag in a psect declaration has been redeclared with a differing. It is not necessary to redeclare this flag.

(1015) missing "*" specification in chipinfo file "*" at line * ***(Driver)***

This attribute was expected to appear at least once but was not defined for this chip.

(1016) missing argument* to "*" specification in chipinfo file "*" at line * ***(Driver)***

This value of this attribute is blank in the chip configuration file.

(1017) extraneous argument* to "*" specification in chipinfo file "*" at line * ***(Driver)***

There are too many attributes for the the listed specification in the chip configuration file.

(1018) illegal number of "*" specification* (* found; * expected) in chipinfo file "*" at line * ***(Driver)***

This attribute was expected to appear a certain number of times but it did not for this chip.

(1019) duplicate "*" specification in chipinfo file "*" at line * ***(Driver)***

This attribute can only be defined once but has been defined more than once for this chip.

(1020) unknown attribute "*" in chipinfo file "*" at line * ***(Driver)***

The chip configuration file contains an attribute that is not understood by this version of the compiler. Has the chip configuration file or the driver been replaced with an equivalent component from another version of this compiler?

(1021) syntax error reading "*" value in chipinfo file "*" at line * ***(Driver)***

The chip configuration file incorrectly defines the specified value for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1022) syntax error reading "*" range in chipinfo file "*" at line * ***(Driver)***

The chip configuration file incorrectly defines the specified range for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1024) syntax error in chipinfo file "*" at line * ***(Driver)***

The chip configuration file contains a syntax error at the line specified.

(1025) unknown architecture in chipinfo file "*" at line * ***(Driver)***

The attribute at the line indicated defines an architecture that is unknown to this compiler.

(1026) missing architecture in chipinfo file "*" at line * ***(Assembler)***

The chipinfo file has a processor section without an ARCH values. The architecture of the processor must be specified. Contact HI-TECH Support if the chipinfo file has not been modified.

(1027) activation was successful ***(Driver)***

The compiler was successfully activated.

(1028) activation was not successful - error code (*) ***(Driver)***

The compiler did not activated successfully.

(1029) compiler not installed correctly - error code (*) ***(Driver)***

This compiler has failed to find any activation information and cannot proceed to execute. The compiler may have been installed incorrectly or incompletely. The error code quoted can help diagnose the reason for this failure. You may be asked for this failure code if contacting HI-TECH Software for assistance with this problem.

(1030) HEXMATE - Intel hex editing utility (Build 1.%i) ***(Hexmate)***

Indicating the version number of the Hexmate being executed.

(1031) USAGE: * [input1.hex] [input2.hex]... [inputN.hex] [options] ***(Hexmate)***

The suggested usage of Hexmate.

(1032) use –HELP=<option> for usage of these command line options ***(Hexmate)***

More detailed information is available for a specific option by passing that option to the HELP option.

(1033) available command-line options: ***(Hexmate)***

This is a simple heading that appears before the list of available options for this application.

(1034) type "*" for available options ***(Hexmate)***

It looks like you need help. This advisory suggests how to get more information about the options available to this application or the usage of these options.

(1035) bad argument count (*) ***(Parser)***

The number of arguments to a function is unreasonable. This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1036) bad "*" optional header length (0x* expected) ***(Cromwell)***

The length of the optional header in this COFF file was of an incorrect length.

(1037) short read on * ***(Cromwell)***

When reading the type of data indicated in this message, it terminated before reaching its specified length.

(1038) string table length too short ***(Cromwell)***

The specified length of the COFF string table is less than the minimum.

(1039) inconsistent symbol count ***(Cromwell)***

The number of symbols in the symbol table has exceeded the number indicated in the COFF header.

(1040) bad checksum: record 0x*, checksum 0x* ***(Cromwell)***

A record of the type specified failed to match its own checksum value.

(1041) short record ***(Cromwell)***

While reading a file, one of the file's records ended short of its specified length.

(1042) unknown * record type 0x* ***(Cromwell)***

The type indicator of this record did not match any valid types for this file format.

(1043) unknown optional header ***(Cromwell)***

When reading this Microchip COFF file, the optional header within the file header was of an incorrect length.

(1044) end of file encountered ***(Cromwell, Linker)***

The end of the file was found while more data was expected. Has this input file been truncated?

(1045) short read on block of * bytes ***(Cromwell)***

A while reading a block of byte data from a UBROF record, the block ended before the expected length.

(1046) short string read ***(Cromwell)***

A while reading a string from a UBROF record, the string ended before the specified length.

(1047) bad type byte for UBROF file ***(Cromwell)***

This UBROF file did not begin with the correct record.

(1048) bad time/date stamp ***(Cromwell)***

This UBROF file has a bad time/date stamp.

(1049) wrong CRC on 0x* bytes; should be * ***(Cromwell)***

An end record has a mismatching CRC value in this UBROF file.

(1050) bad date in 0x52 record ***(Cromwell)***

A debug record has a bad date component in this UBROF file.

(1051) bad date in 0x01 record ***(Cromwell)***

A start of program record or segment record has a bad date component in this UBROF file.

(1052) unknown record type ***(Cromwell)***

A record type could not be determined when reading this UBROF file.

(1058) assertion ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1059) rewrite loop ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1081) static initialization of persistent variable "*" ***(Parser, Code Generator)***

A persistent variable has been assigned an initial value. This is somewhat contradictory as the initial value will be assigned to the variable during execution of the compiler's startup code, however the *persistent* qualifier requests that this variable shall be unchanged by the compiler's startup code.

(1082) size of initialized array element is zero ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1088) function pointer "*" is used but never assigned a value ***(Code Generator)***

A function call involving a function pointer was made, but the pointer was never assigned a target address, e.g.:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1089) recursive function call to "*" ***(Code Generator)***

A recursive call to the specified function has been found. The call may be direct or indirect (using function pointers) and may be either a function calling itself, or calling another function whose call graph includes the function under consideration.

(1090) variable "*" is not used *(Code Generator)*

This variable is declared but has not been used by the program. Consider removing it from the program.

(1091) main function "*" not defined *(Code Generator)*

The *main* function has not been defined. Every C program must have a function called *main*.

(1094) bad derived type *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1095) bad call to typeSub() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1096) type should be unqualified *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1097) unknown type string "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1098) conflicting declarations for variable "*" (*:*) *(Parser, Code Generator)*

Differing type information has been detected in the declarations for a variable, or between a declaratin and the definition of a variable, e.g.:

```
extern long int test;
int test;    /* oops -- which is right? int or long int ? */
```

(1104) unqualified error *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1118) bad string "*" in getexpr(J) *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1119) bad string "*" in getexpr(LRN) ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1121) expression error ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1137) match() error: * ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1157) W register must be W9 ***(Assembler)***

The working register required here has to be W9, but an other working register was selected.

(1159) W register must be W11 ***(Assembler)***

The working register required here has to be W11, but an other working register was selected.

(1178) the "*" option has been removed and has no effect ***(Driver)***

This option no longer exists in this version of the compiler and has been ignored. Use the compiler's *–help* option or refer to the manual to find a replacement option.

(1179) interrupt level for function "*" may not exceed * ***(Code Generator)***

The interrupt level for the function specified is too high. Each interrupt function is assigned a unique interrupt level. This level is considered when analysing the call graph and re-entrantly called functions. If using the `interrupt_level` pragma, check the value specified.

(1180) directory "*" does not exist ***(Driver)***

The directory specified in the setup option does not exist. Create the directory and try again.

(1182) near variables must be global or static ***(Code Generator)***

A variable qualified as *near* must also be qualified with *static* or made global. An auto variable cannot be qualified as *near*.

(1183) invalid version number *(Activation)*

During activation, no matching version number was found on the HI-TECH activation server database for the serial number specified.

(1184) activation limit reached *(Activation)*

The number of activations of the serial number specified has exceeded the maximum number allowed for the license.

(1185) invalid serial number *(Activation)*

During activation, no matching serial number was found on the HI-TECH activation server database.

(1186) licence has expired *(Driver)*

The time-limited license for this compiler has expired.

(1187) invalid activation request *(Driver)*

The compiler has not been correctly activated.

(1188) network error * *(Activation)*

The compiler activation software was unable to connect to the HI-TECH activation server via the network.

(1190) FAE license only - not for use in commercial applications *(Driver)*

Indicates that this compiler has been activated with an FAE licence. This licence does not permit the product to be used for the development of commercial applications.

(1191) licensed for educational use only *(Driver)*

Indicates that this compiler has been activated with an education licence. The educational licence is only available to educational facilities and does not permit the product to be used for the development of commercial applications.

(1192) licensed for evaluation purposes only *(Driver)*

Indicates that this compiler has been activated with an evaluation licence.

(1193) this licence will expire on * *(Driver)*

The compiler has been installed as a time-limited trial. This trial will end on the date specified.

(1195) invalid syntax for "*" option *(Driver)*

A command line option that accepts additional parameters was given inappropriate data or insufficient data. For example an option may expect two parameters with both being integers. Passing a string as one of these parameters or supplying only one parameter could result in this error.

(1198) too many "*" specifications; * maximum *(Hexmate)*

This option has been specified too many times. If possible, try performing these operations over several command lines.

(1199) compiler has not been activated *(Driver)*

The trial period for this compiler has expired. The compiler is now inoperable until activated with a valid serial number. Contact HI-TECH Software to purchase this software and obtain a serial number.

(1200) Found %0*lXh at address *h *(Hexmate)*

The code sequence specified in a -FIND option has been found at this address.

(1201) all FIND/REPLACE code specifications must be of equal width *(Hexmate)*

All find, replace and mask attributes in this option must be of the same byte width. Check the parameters supplied to this option. For example finding 1234h (2 bytes) masked with FFh (1 byte) will result in an error, but masking with 00FFh (2 bytes) will be Ok.

(1202) unknown format requested in -FORMAT: * *(Hexmate)*

An unknown or unsupported INHX format has been requested. Refer to documentation for supported INHX formats.

(1203) unpaired nibble in * value will be truncated *(Hexmate)*

Data to this option was not entered as whole bytes. Perhaps the data was incomplete or a leading zero was omitted. For example the value Fh contains only four bits of significant data and is not a whole byte. The value 0Fh contains eight bits of significant data and is a whole byte.

(1204) * value must be between 1 and * bytes long ***(Hexmate)***

An illegal length of data was given to this option. The value provided to this option exceeds the maximum or minimum bounds required by this option.

(1205) using the configuration file *; you may override this with the environment variable HTC_XML ***(Driver)***

This is the compiler configuration file selected during compiler setup. This can be changed via the HTC_XML environment variable. This file is used to determine where the compiler has been installed.

(1207) some of the command line options you are using are now obsolete ***(Driver)***

Some of the command line options passed to the driver have now been discontinued in this version of the compiler, however during a grace period these old options will still be processed by the driver.

(1208) use –help option or refer to the user manual for option details ***(Driver)***

An obsolete option was detected. Use –help or refer to the manual to find a replacement option that will not result in this advisory message.

(1209) An old MPLAB tool suite plug-in was detected. ***(Driver)***

The options passed to the driver resemble those that the Microchip MPLAB IDE would pass to a previous version of this compiler. Some of these options are now obsolete, however they were still interpreted. It is recommended that you install an updated HI-TECH options plug-in for the MPLAB IDE.

(1210) Visit the HI-TECH Software website (www.htsoft.com) for a possible update ***(Driver)***

Visit our website to see if an update is available to address the issue(s) listed in the previous compiler message. Please refer to the on-line self-help facilities such as the *Frequently asked Questions* or search the *On-line forums*. In the event of no details being found here, contact HI-TECH Software for further information.

(1211) Memory type "*" is not valid for this device ***(Driver)***

A command-line option attempted to add a type of memory to this device that is not supported by this device. For example, adding external RAM to a device that does not have an external memory interface.

(1212) Found * (%0*lXh) at address *h ***(Hexmate)***

The code sequence specified in a -FIND option has been found at this address.

(1213) duplicate ARCH for * in chipinfo file at line * ***(Assembler, Driver)***

The chipinfo file has a processor section with multiple ARCH values. Only one ARCH value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(1214) duplicate RAMSIZE for * in chipinfo file at line * ***(Assembler)***

More than one RAMSIZE entry was found in the chipinfo file for this particular chip.

(1215) can't open chipinfo file * ***(Assembler)***

The chip configuration file was not able to be opened for reading. Check that the file's pathname is correct.

(1216) empty chipinfo file * ***(Assembler)***

The chip configuration file was able to be opened but was found to be empty. This file may have been corrupted.

(1217) can't open command file * ***(Assembler)***

The input command file could not be opened for reading. Check that the file's pathname is correct.

(1218) can't create cross reference file * ***(Assembler)***

The assembler attempted to create a cross reference file, but it could not be created. Check that the file's pathname is correct.

(1219) can't create list file * ***(Assembler)***

The assembler could not open or create an assembler listing file. Check that the file's pathname is correct. Is the file attempting to be created in a read-only directory or is the file already open in another application?

(1220) can't create assembler file * ***(Assembler)***

The assembler could not open or create an assembler output file. Check that the file's pathname is correct. Is the file attempting to be created in a read-only directory or is the file already open in another application?

(1221) can't create relocatable list file * ***(Assembler)***

The assembler could not open or create its relocatable list file. Is the file attempting to be created in a read-only directory or is the file already open in another application?

(1222) can't create object file * ***(Assembler)***

The assembler could not open or create its output object file. Check that the file's pathname is correct. Is the file attempting to be created in a read-only directory or is the file already open in another application?

(1223) relative branch/call offset out of range ***(Assembler)***

The destination of a relative branch or call instruction was too far away for the instruction to reach. These instructions have a limited reach. Try using an instruction other than a relative branch/call to get to the destination, or bring the destination closer.

(1224) banked/common conflict ***(Assembler)***

The assembler has found conflicting information that suggests that a symbol is located in the access bank, but also in the banked RAM area, e.g.:

```
movwf  c:_foo,b  ; _foo cannot be common and banked
```

(1225) LFSR instruction argument must be 0-3 ***(Assembler)***

The LFSR instruction's first parameter must be within the range 0 to 3.

(1228) unable to locate installation directory ***(Driver)***

The compiler cannot determine the directory where it has been installed.

(1230) dereferencing uninitialized pointer "*" ***(Code Generator)***

A pointer that has not yet been assigned a value has been dereferenced. This can result in erroneous behaviour at runtime.

(1232) persistent data may be corrupted during asynchronous reset (see errata) ***(Driver)***

For some PIC18 chips, data may become corrupted during the event of an asynchronous reset. Refer to the Microchip errata document for more details about how this chip is affected. This

(1233) Employing * errata work-arounds: ***(Driver)***

The compiler is applying software workarounds for known issues in the selected device. Consult the errata document for this device to see whether it is safe to disable the compiler's workaround for any of the listed problems.

(1234) * * ***(Driver)***

Listing a silicon defect that the compiler is working around. Software workarounds generally increase the overall code size. Refer to the errata document for the device you are using to see whether the defect affects your program. If not, you may save space by disabling the workaround.

(1235) unknown keyword * ***(Driver)***

The token contained in the USB descriptor file was not recognised.

(1236) invalid argument to *: * ***(Driver)***

An option that can take additional parameters was given an invalid parameter value. Check the usage of the option or the syntax or range of the expected parameter.

(1237) endpoint 0 is pre-defined ***(Driver)***

An attempt has been made to define endpoint 0 in a USB file. This channel c

(1238) FNALIGN failure on * ***(Linker)***

Two functions have their auto/parameter blocks aligned using the `FNALIGN` directive, but one function calls the other, which implies that must not be aligned. This will occur if a function pointer is assigned the address of each function, but one function calls the other. For example:

```
int one(int a) { return a; }
int two(int a) { return two(a)+2; }  /* ! */
int (*ip)(int);
ip = one;
ip(23);
```

```
ip = two;       /* ip references one and two; two calls one */
ip(67);
```

(1239) pointer * has no valid targets ***(Code Generator)***

A function call involving a function pointer was made, but the pointer was never assigned a target address, e.g.:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1240) unknown checksum algorithm type (%i) ***(Driver)***

The error file specified after the `-Efile` or `-E+file` options could not be opened. Check to ensure that the file or directory is valid and that has read only access.

(1241) bad start address in * ***(Driver)***

The start of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1242) bad end address in * ***(Driver)***

The end of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1243) bad destination address in * ***(Driver)***

The destination address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1245) value greater than zero required for * ***(Hexmate)***

The *align* operand to the HEXMATE `-FIND` option must be positive.

(1246) no RAM defined for variable placement ***(Code Generator)***

No memory has been specified to cover the banked RAM memory.

(1247) no access RAM defined for variable placement ***(Code Generator)***

No memory has been specified to cover the access bank memory.

(1248) symbol (*) encountered with undefined type size ***(Code Generator)***

The code generator was asked to position a variable, but the size of the variable is not known. This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1249) could not find space (* byte*) for variable * in access bank ***(Code Generator)***

The code generator could not find space in the access bank RAM for the variable specified. Variables qualifed as `near` are forced into this area of memory.

(1250) could not find space (* byte*) for variable * ***(Code Generator)***

The code generator could not find space in the banked RAM for the variable specified.

(1251) no far RAM defined for variable placement ***(Code Generator)***

Variables were qualified as `far` in the source code, but no memory has been specified to hold these objects. Variables qualfiied as `far` will reside in the program space memory, but are writable. Memory can be secified using the `--RAM` option with address ranges above the top of the on-chip program space memory.

(1252) could not find space (* byte*) for variable * in far RAM ***(Code Generator)***

The code generator could not find space in RAM for the psect that holds variables qualified as `far`.

(1253) could not find space (* byte*) for auto/param block ***(Code Generator)***

The code generator could not find space in RAM for the psect that holds `auto` and parameter variables.

(1254) could not find space (* byte*) for data block ***(Code Generator)***

The code generator could not find space in RAM for the data psect that holds initialised variables.

(1255) conflicting paths for output directory *(Driver)*

The compiler has been given contradictory paths for the output directory via any of the `-O` or `--OUTDIR` options, e.g.

```
--outdir=../../   -o../main.hex
```

(1256) undefined symbol "*" treated as hex constant *(Assembler)*

A token which could either be interpreted as a symbol or a hexadecimal value does not match any previously defined symbol and so will be interpreted as the latter. Use a leading *zero* to avoid the ambiguity, or use an alternate radix sepcifier such as `0x`. For example:

```
mov  a, F7h  ; is this the symbol F7h, or the hex number 0xF7?
```

(1257) local variable "*" is used but never given a value *(Code Generator)*

An `auto` variable has been defined and used in an expression, but it has not been assigned a value in the C code before its first use. Auto variables are not cleared on startup and their initial value is undefined. For example:

```
void main(void) {
  double src, out;
  out = sin(src);   /* oops -- what value was in src? */
```

(1258) possible stack overflow when calling function "*" *(Code Generator)*

The call tree analysis by the code generator indicates that the hardware stack may overflow. This should be treated as a guide only. Interrupts, the assembler optimizer and the program structure may affect the stack usage. The stack usuage is based on the C program and does not include any call tree derived from assembly code.

(1259) can't optimize for both speed and space *(Driver)*

The driver has been given contradictory options of compile for speed and compile for space, e.g.

```
--opt=speed,space
```

(1260) macro "*" redefined *(Assembler)*

More than one definition for a macro with the same name has been encountered, e.g.

```
MACRO fin
  ret
ENDM
MACRO fin   ; oops -- was this meant to be a different macro?
  reti
ENDM
```

(1261) string constant required *(Assembler)*

A string argument is required with the `DS` or `DSU` directive, e.g.

```
DS ONE     ; oops -- did you mean DS "ONE"?
```

(1264) unsafe pointer conversion *(Code Generator)*

A pointer to one kind of structure has been converted to another kind of structure and the structures do not have a similar definition, e.g.

```
struct ONE {
  unsigned a;
  long b;          /* ! */
} one;
struct TWO {
  unsigned a;
  unsigned b;      /* ! */
} two;
struct ONE * oneptr;
oneptr = & two;    /* oops --
                   was ONE meant to be same struct as TWO? */
```

(1267) fixup overflow referencing * * (0x*) into * byte* at 0x* (*** */0x*) *(Linker)*

See the following error message (1268) for more information..

(1268) fixup overflow storing 0x* in * byte* at 0x* (* */0x*)** ***(Linker)***

Fixup is the process conducted by the linker of replacing symbolic references to variables etc, in an assembler instruction with an absolute value. This takes place after positioning the psects (program sections or blocks) into the available memory on the target device. Fixup overflow is when the value determined for a symbol is too large to fit within the allocated space within the assembler instruction. For example, if an assembler instruction has an 8-bit field to hold an address and the linker determines that the symbol that has been used to represent this address has the value 0x110, then clearly this value cannot be inserted into the instruction.

(1269) there * * day* left until this licence will expire ***(Driver)***

This compiler has not been activated and is running as a demo. The time indicated is how long the demo period will continue.

(1273) Omniscient Code Generation not available in Lite mode ***(Driver)***

When running in Lite mode, the advanced Omniscient Code Generation (OCG) features are disabled. This will result in much larger code than would be produced when running in PRO mode.

(1274) delay exceeds maximum limit of * cycles ***(Code Generator)***

The argument to the in-line delay routine (`_delay`) is limited to the maximum size indicated. Use the routine consequtively, or place it is a loop to acheive the desired delay period.

(1282) no REAL ICE transport options specified ***(Driver)***

When selecting the Microchip MPLAB REAL ICE as the debugger, the `--debugger` option must include the transport type for trace facilities.

(1283) illegal table pointer address size * (__activetblptr) ***(Driver)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1284) malformed mapfile while generating summary: CLASS expected but not found*(Driver)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1285) malformed mapfile while generating summary: no name at position * ***(Driver)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1286) malformed mapfile while generating summary: no link address at position * *(Driver)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1287) malformed mapfile while generating summary: no load address at position * *(Driver)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1288) malformed mapfile while generating summary: no length at position * *(Driver)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1289) line range limit exceeded, debugging may be affected *(Cromwell)*

Internally Cromwell can only handle a limited number of addresses which correspond to a single line of C code. In all but the most perverse cases this limit shouldn't be reached. However if it has then consider breaking up the related C statement into a series of simpiler statements. If that is not possible or successful then contact HI-TECH Software technical support with details.

(1290) DWARF: Buffer overflow in DIE *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1291) bad ELF string table index *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1292) malformed define in .SDB file * *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1293) DWARF: couldn't find type for * *(Cromwell)*

This is an internal compiler warning. Contact HI-TECH Software technical support with details.

(1294) there is only one day left until this licence expires *(Driver)*

The compiler was fully activated for a limited evaluation period. That period is about to expire in one day. When expired the compiler will switch to Lite mode - if available - or cease to function. To fully reactivate the compiler a valid serial number is required. Please contact our sales department for more details.

(1295) there are * days left until this licence will expire ***(Driver)***

The compiler was fully activated for a limited evaluation period. That period expire soon. When expired the compiler will switch to Lite mode - if available - or cease to function. To fully reactivate the compiler a valid serial number is required. Please contact our sales department for more details.

(1296) source file "*" conflicts with "*" ***(Driver)***

If two source files with the same name but different paths are given as arguments to the driver this fatal error will occur. For example foobar.c may be specified twice but in different directories. This is illegal in the cases where the compiler or IDE assumes that foobar.c will produce an intermediate file foobar.p1 and all intermediate files are output in the same directory regardless of the source directory. Consider renaming one of these files.

(1297) option * not available in Lite mode ***(Driver)***

Although fully functional, compilers in Lite mode have certain driver options disabled. These options are those (such as as –setoption and –getoption) which would allow users to circumvent the restrictions on optimisations.

(1298) use of * outside macros is illegal ***(Assembler)***

Some assembler directives, e.g. `EXITM`, can only be used inside macro definitions.

(1299) non-standard modifier "*" - use "*" instead ***(Parser)***

A printf placeholder modifier has been used which is non-standard. Use the indicated modifier instead. For example, the standard `hh` modifier should be used in preference to b to indicate that the value should be printed as a `char` type.

(1301) invalid ELF section header. Skipping ***(Cromwell)***

Cromwell found an invalid section in an ELF section header. This section will be skipped.

(1302) could not find valid ELF output extension for this device ***(Cromwell)***

The extension could not be for the target device family.

(1303) invalid variable location detected: * - * ***(Cromwell)***

A symbol location could not be determined from the SDB file.

(1304) unknown register name: "*" ***(Cromwell)***

The location for the indicated symbol in the SDB file was a register, but the register name was not recognized.

(1305) inconsistent storage class for variable: "*" ***(Cromwell)***

The storage class for the indicated symbol in the SDB file was not recognized.

(1306) inconsistent size (* vs *) for variable: "*" ***(Cromwell)***

The size of the symbol indicated in the SDB file does not match the size of its type.

(1307) psect * truncated to * bytes ***(Driver)***

The psect representing either the stack or heap could not be made as large as requested and will be truncated to fit the available memory space.

(1309) ignoring invalid runtime * sub-option (*) using default ***(Driver)***

The indicated suboption to the `--RUNTIME` option is malformed, e.g.

```
--RUNTIME=default,speed:0y1234
```

Oops, that should be `0x1234`.

(1310) specified speed (*Hz) exceeds max operating frequency (*Hz), defaulting to *Hz ***(Driver)***

The frequency specified to the perform suboption to `--RUNTIME` option is too large for the selected device.

```
--RUNTIME=default,speed:0xffffffff
```

Oops, that value is too large.

(1311) missing configuration setting for config word *, using default ***(Driver)***

The configuration settings for the indicated word have not be supplied in the source code and a default value will be used.

(1312) conflicting runtime perform sub-option and configuration word settings, assuming *Hz ***(Driver)***

The configuration settings and the value specified with the perform suboption of the `--RUNTIME` options conflict and a default frequency has been selected.

(1313) * sub-options ("*") ignored ***(Driver)***

The argument to a suboption is not required and will be ignored.

```
--OUTPUT=intel:8
```

Oops, the `:8` is not required.

(1314) illegal action in memory allocation ***(Code Generator)***

This is an internal error. Contact HI-TECH Support with details.

(1315) undefined or empty class used to link psect * ***(Linker)***

The linker was asked to place a psect within the range of addresses specified by a class, but the class was either never defined, or contains no memory ranges.

(1316) attribute "*" ignored ***(Parser)***

An attribute has been encountered that is valid, but which is not implemented by the parser. It will be ignored by the parser and the attribute will have no effect. Contact HI-TECH Support with details.

(1317) missing argument to attribute "*" ***(Parser)***

An attribute has been encountered that requires an argument, but this is not present. Contact HI-TECH Support with details.

(1318) invalid argument to attribute "*" ***(Parser)***

An argument to an attribute has been encountered, but it is malformed. Contact HI-TECH Support with details.

(1319) invalid type "*" for attribute "*" ***(Parser)***

This indicated a bad option passed to the parser. Contact HI-TECH Support with details.

(1320) attribute "*" already exists ***(Parser)***

This indicated the same attribute option being passed to the parser more than once. Contact HI-TECH Support with details.

(1321) bad attribute -T option "%s" ***(Parser)***

The attribute option passed to the parser is malformed. Contact HI-TECH Support with details.

(1322) unknown qualifier "%s" given to -T ***(Parser)***

The qualifier specified in an attribute option is not known. Contact HI-TECH Support with details.

(1323) attribute expected ***(Parser)***

The __attribute__ directive was used but did not specify an attribute type.

```
int rv (int a) __attribute__(()) /* oops -- what is the attribute? */
```

(1324) qualifier "*" ignored ***(Parser)***

Some qualifiers are valid, but may not be implemented on some compilers or target devices. This warning indicates that the qualifier will be ignored.

(1327) interrupt function "*" redefined by "*" ***(Code Generator)***

An interrupt function has been written that is linked to a vector location that already has an interrupt function lined to it.

```
void interrupt timer1_isr(void) @ TIMER_1_VCTR
{ ... }
void interrupt timer2_isr(void) @ TIMER_1_VCTR
{ ... } /* oops -- did you mean that to be TIMER_2_VCTR */
```

(1342) whitespace after "\" ***(Preprocessor)***

Whitespace characters have been found between a backslash and newline characters and will be ignored.

(1343) hexfile data at address 0x* (0x*) overwritten with 0x* ***(Objtohex)***

The indicated address is about to be overwritten by additional data. This would indicate more than one section of code contributing to the same address.

(1346) can't find 0x* words for psect "*" in segment "*" (largest unused contiguous range 0x%lX) ***(Linker)***

See also message (491). The new form of message also indicates the largest free block that the linker could find. Unless there is a single space large enough to accommodate the psect, the linker will issue this message. Often when there is banking or paging involved the largest free space is much smaller than the total amount of space remaining.

(1347) can't find 0x* words (0x* withtotal) for psect "*" in segment "*" (largest unused contiguous range 0x%lX) ***(Linker)***

See also message (593). The new form of message also indicates the largest free block that the linker could find. Unless there is a single space large enough to accommodate the psect, the linker will issue this message. Often when there is banking or paging involved the largest free space is much smaller than the total amount of space remaining.

(1348) enum tag "*" redefined (from *:*) ***(Parser)***

More than one `enum` tag with the same name has been defined, The previous definition is indicated in the message.

```
enum VALS { ONE=1, TWO, THREE };
enum VALS { NINE=9, TEN }; /* oops -- is INPUT the right tag name? */
```

(1350) pointer operands to "-" must reference the same array ***(Code Generator)***

If two addresses are subtracted, the addresses must be of the same object to be ANSI compliant.

```
int * ip;
int fred, buf[20];
ip = &buf[0] - &fred;  // oops --second operand must be an address of
                       // a "buf" element
```

(1352) truncation of operand value (0x*) to * bits ***(Assembler)***

The operand to an assembler instruction was too large and was truncated.

```
movlw 0x321 ; oops -- is this the right value?
```

(1354) ignoring configuration setting for unimplemented word * ***(Driver)***

A configuration word setting was specified for a word that does not exist on the target device.

```
__CONFIG(3, 0x1234); /* config word 3 does not exist on an 18C801 */
```

(1355) inline delay argument too large ***(Code Generator)***

The inline delay sequence `_delay` has been used, but the number of instruction cycles requested is too large. Use this routine multiple times to achieve the desired delay length.

```
#include <htc.h>
void main(void) {
   delay(0x400000); /* oops -- cannot delay by this number of cycles */
}
```

(1356) fixup overflow referencing * * (0x*) into * byte* at 0x*/0x* -> 0x* (* */0x*)** ***(Linker)***

See also message (477). This form of the message calculates the address of the offending instruction taking into account the delta value of the psect which contains the instruction.

(1357) fixup overflow storing 0x* in * byte* at 0x*/0x* -> 0x* (* */0x*)** ***(Linker)***

See also message (477). This form of the message calculates the address of the offending instruction taking into account the delta value of the psect which contains the instruction.

(1358) no space for * temps (*) ***(Code Generator)***

The code generator was unable to find a space large enough to hold the temporary variables (scratch variables) for this program.

(1359) no space for * parameters ***(Code Generator)***

The code generator was unable to find a space large enough to hold the parameter variables for a particular function.

(1360) no space for auto/param * ***(Code Generator)***

The code generator was unable to find a space large enough to hold the `auto` variables for a particular function. Some parameters passed in registers may need to be allocated space in this `auto` area as well.

(1361) syntax error in configuration argument ***(Parser)***

The argument to `#pragma config` was malformed.

```
#pragma config WDT  /* oops -- is WDT on or off? */
```

(1362) configuration setting *=* redefined ***(Code Generator)***

The same `config` pragma setting has been issued more than once with different values.

```
#pragma config WDT=OFF
#pragma config WDT=ON  /* oops -- is WDT on or off? */
```

(1363) unknown configuration setting (* = *) used ***(Driver)***

The configuration value and setting is not known for the target device.

```
#pragma config WDR=ON /* oops -- did you mean WDT? */
```

(1364) can't open configuration registers data file * ***(Driver)***

The file containing value configuration settings could not be found.

(1365) missing argument to pragma "varlocate" ***(Parser)***

The argument to `#pragma varlocate` was malformed.

```
#pragma varlocate /* oops -- what do you want to locate & where? */
```

(1366) syntax error in pragma "varlocate" ***(Parser)***

The argument to `#pragma varlocate` was malformed.

```
#pragma varlocate fred /* oops -- which bank for fred? */
```

(1367) end of file in _asm ***(Parser)***

An end-of-file marker was encountered inside a `_asm _endasm` block.

(1368) assembler message: * ***(Assembler)***

Displayed is an assembler advisory message produced by the MESSG directive contained in the assembler source.

(1369) can't open proc file * ***(Driver)***

The proc file for the selected device could not be opened.

(1371) float type can't be bigger then double type; double has been changed to * bits ***(Driver)***

Use of the `--float` and `--double` options has result in the size of the `double` type being smaller than that of the `float` type. This is not permitted by the C Standard. The `double` type size has been increased to be that indicated.

(1375) multiple interrupt functions (* and *) defined for device with only one interrupt vector ***(Code Generator)***

The named functions have both been qualified `interrupt`, but the target device only supports one interrupt vector and hence one interrupt function.

```
interrupt void isr_lo(void) {
   // ...
}
interrupt void isr_hi(void) {
   // ...
}
```

(1376) initial value (*) too large for bitfield width (*) ***(Code Generator)***

A structure with bit-fields has been defined an initialized with values. The value indicated it too large to fit in the corresponding bitfield width.

```
struct {
   unsigned flag :1;
   unsigned mode :3;
}
foobar={1,100};  //oops, 100 is too large for a 3-bit wide object
```

(1377) no suitable strategy for this switch ***(Code Generator)***

The compiler was unable to determine the switch strategy to use to encode a C `switch` statement based on the code and your selection using the `#pragma switch` directive. You may need to choose a different strategy.

(1387) inline delay argument must be constant ***(Code Generator)***

The `__delay` inline function can only take a constant expression as its argument.

```
int delay_val = 99;
__delay(delay_val); // oops, argument must be a constant expression
```

(1390) identifier specifies insignificant characters beyond maximum identifier length ***(Parser)***

An identifier has been used that is so long that it exceeds the set identifier length. This may mean that long identifiers may not be correctly identified and the code will fail. The maximum identifier length can be adjusted using the `-N` option.

```
int theValueOfThePortAfterTheModeBitsHaveBeenSet; // oops,
                  // make your symbol shorter or increase the maximum
                  // identifier length
```

(1393) possible hardware stack overflow detected, estimated stack depth: * ***(Code Generator)***

The compiler has detected that the call graph for a program may be using more stack space that allocated on the target device. If this is the case, the code may fail. The compiler can only make assumptions regarding the stack usage when interrupts are involved and these lead to a worst-case estimate of stack usage. Confirm the function call nesting if this warning is issued.

(1394) attempting to create memory range (* - *) larger than page size, * ***(Driver)***

The compiler driver has detected that the memory settings include a program memory "page" that is larger than the page size for the device. This would mostly likely be the case if the `--ROM` option is used to change the default memory settings. Consult you device data sheet to determine the page size of the device you are using and ensure that any contiguous memory range you specify using the `--ROM` option has a boundary that corresponds to the device page boundaries.

```
--ROM=100-1fff
```

The above may need to be paged. If the page size is 800h, the above could specified as

```
--ROM=100-7ff,800-fff,1000-17ff,1800-1fff
```

(1395) notable code sequence candidate suitable for compiler validation suite detected (*) ***(Code Generator)***

The compiler has in-built checks that can determine if combinations of internal code templates have been encountered. Where unique combinations are uncovered when compiling code, this message is issued. This message is not an error or warning, and its presence does not indicate possible code failure, but if you are willing to participate, the code you are compiling can be sent to Support to assist with the compiler testing process.

(1396) "*" positioned in the * memory region (0x* - 0x*) reserved by the compiler ***(Code Generator)***

Some memory regions are reserved for use by the compiler. These regions are not normally used to allocate variables defined in your code. However, by making variables absolute, it is possible to place variables in these regions and avoid errors that would normally be issued by the linker. (Absolute variables can be placed at any location, even on top of other objects.) This warning from the code generator indicates that an absolute has been detected that will be located at memory that the compiler will be reserving. You must locate the absolute variable at a different location. This message will commonly be issued when placing variables in the common memory space.

```
char shared @ 0x7; // oops, this memory is required by the compiler
```

(1397) unable to implement non-stack call to "*"; possible hardware stack overflow ***(Code Generator)***

The compiler must encode a C function call without using a CALL assembly instruction and the hardware stack (and instead use a lookup table), but is unable to. A call might be required if the function is called indirectly via a pointer, but if the hardware stack is already full, an additional call will cause a stack overflow.

(1401) eeprom qualified variables cannot be accessed from both interrupt and mainline code ***(Code Generator)***

All eeprom variables are accessed via routines which are not reentrant. Code might fail if an attempt is made to access eeprom variables from interrupt and main-line code. Avoid accessing eeprom variables in interrupt functions.

(1402) a pointer to eeprom cannot also point to other data types ***(Code Generator)***

A pointer cannot have targets in both the eeprom space and ordinary data space.

(1404) unsupported: * ***(Parser)***

The unsupported __attribute__ has been used to indicate that some code feature is not supported. The message printed will indicate the feature that is not supported.

(1406) auto eeprom variables are not supported ***(Code Generator)***

Variables qualified as `eeprom` cannot be `auto`. You can define `static` local objects qualified as `eeprom`, if required.

(1407) bit eeprom variables are not supported ***(Code Generator)***

Variables qualified as `eeprom` cannot have type `bit`.

(1408) ignoring initialization of far variables ***(Code Generator)***

Variables qualified as `far` cannot be assigned an intial value. Assign the value later in the code.

```
far int chan = 0x1234; // oops -- you can't assign a value here
```

(1409) Warning number used with pragma "warning" is invalid ***(Parser)***

The message number usd with the `warning` pragma is below zero or larger than the highest mesasge number available.

```
#pragma warning disable 1316 13350 // oops -- maybe number 1335?
```

(1410) Cannot assign the result of an invalid function pointer ***(Code Generator)***

The compiler will allow some functions to be called via a constant cast to be a function pointer, but not all. The address specified is not valid for this device.

```
foobar += ((int (*)(int))0x0)(77);
    // oops -- you cannot call a function with a NULL pointer
```

(1411) Additional ROM range out of bounds ***(Driver)***

Program memory specified with the `--ROM` option is outside of the on-chip, or external, memory range supported by this device.

(1412) missing argument to pragma "warning disable" ***(Parser)***

Following the `#pragma warning disable` should be a comma-separated list of message numbers to disable.

```
#pragma warning disable // oops -- what messages are to be disabled?
```

Try something like the folllwing.

```
#pragma warning disable 1362
```

(0) delete what ? ***(Libr)***

The librarian requires one or more modules to be listed for deletion when using the `d` key, e.g.:

```
libr d c:\ht-pic\lib\pic704-c.lib
```

does not indicate which modules to delete. try something like:

```
libr d c:\ht-pic\lib\pic704-c.lib wdiv.obj
```

(0) incomplete ident record ***(Libr)***

The IDENT record in the object file was incomplete. Contact HI-TECH Support with details.

(0) incomplete symbol record ***(Libr)***

The SYM record in the object file was incomplete. Contact HI-TECH Support with details.

(0) library file names should have .lib extension: * ***(Libr)***

Use the `.lib` extension when specifying a library filename.

(0) module * defines no symbols ***(Libr)***

No symbols were found in the module's object file. This may be what was intended, or it may mean that part of the code was inadvertently removed or commented.

(0) replace what ? ***(Libr)***

The librarian requires one or more modules to be listed for replacement when using the `r` key, e.g.:

```
libr r lcd.lib
```

This command needs the name of a module (`.obj` file) after the library name.

Appendix C

Chip Information

The following table lists all devices currently supported by HI-TECH C Compiler for PIC18 MCUs.

Table C.1: Devices supported by HI-TECH C Compiler for PIC18 MCUs

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18C242	4000	200		
18C252	8000	600		
18C442	4000	200		
18C452	8000	600		
18C601	0	600		0-1FFFFF
18C658	8000	600		
18C801	0	600		0-1FFFFF
18C858	8000	600		
18F1220	1000	100		
18F1230	1000	100		
18F1320	2000	100		
18F1330	2000	100		
18F13K22	2000	100		
18F13K50	2000	200		
18F14K22	4000	200		
18F14K22LIN	4000	200		
18F14K50	4000	300		
18F2220	1000	200		
18F2221	1000	200		
18F2320	2000	200		
18F2321	2000	200		
18F2331	2000	300		
18F23K20	2000	200		
18F23K22	2000	200		

continued...

Table C.1: Devices supported by HI-TECH C Compiler for PIC18 MCUs

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F2410	4000	300		
18F242	4000	300		
18F2420	4000	300		
18F2423	4000	300		
18F2431	4000	300		
18F2439	3000	280		
18F2450	4000	300		
18F2455	6000	800		
18F2458	6000	800		
18F248	4000	300		
18F2480	4000	300		
18F24J10	3FF8	400		20000-1FFFFF
18F24J11	3FF8	EC0		
18F24J50	3FF8	EC0		
18F24K20	4000	300		
18F24K22	4000	300		
18F2510	8000	600		
18F2515	C000	F80		
18F252	8000	600		
18F2520	8000	600		
18F2523	8000	600		
18F2525	C000	F80		
18F2539	6000	580		
18F2550	8000	800		
18F2553	8000	800		
18F258	8000	600		
18F2580	8000	600		
18F2585	C000	D00		
18F25J10	7FF8	400		20000-1FFFFF
18F25J11	7FF8	EC0		
18F25J50	7FF8	EC0		
18F25K20	8000	600		
18F25K22	8000	600		
18F25K80	8000	E41		
18F2610	10000	F80		
18F2620	10000	F80		
18F2680	10000	D00		
18F2682	14000	D00		
18F2685	18000	D00		
18F26J11	FFF8	EC0		
18F26J13	FFF8	EB0		
18F26J50	FFF8	EC0		
18F26J53	FFF8	EB0		
18F26K20	10000	F60		
18F26K22	10000	F38		

continued...

Table C.1: Devices supported by HI-TECH C Compiler for PIC18 MCUs

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F26K80	10000	E41		
18F27J13	1FFF8	EB0		
18F27J53	1FFF8	EB0		
18F4220	1000	200		
18F4221	1000	200		
18F4320	2000	200		
18F4321	2000	200		
18F4331	2000	300		
18F43K20	2000	200		
18F43K22	2000	200		
18F4410	4000	300		
18F442	4000	300		
18F4420	4000	300		
18F4423	4000	300		
18F4431	4000	300		
18F4439	3000	280		
18F4450	4000	300		
18F4455	6000	800		
18F4458	6000	800		
18F448	4000	300		
18F4480	4000	300		
18F44J10	3FF8	400		20000-1FFFFF
18F44J11	3FF8	EC0		
18F44J50	3FF8	EC0		
18F44K20	4000	300		
18F44K22	4000	300		
18F4510	8000	600		
18F4515	C000	F80		
18F452	8000	600		
18F4520	8000	600		
18F4523	8000	600		
18F4525	C000	F80		
18F4539	6000	580		
18F4550	8000	800		
18F4553	8000	800		
18F458	8000	600		
18F4580	8000	600		
18F4585	C000	D00		
18F45J10	7FF8	400		20000-1FFFFF
18F45J11	7FF8	EC0		
18F45J50	7FF8	EC0		
18F45K20	8000	600		
18F45K22	8000	600		
18F45K80	8000	E41		
18F4610	10000	F80		

continued...

Table C.1: Devices supported by HI-TECH C Compiler for PIC18 MCUs

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F4620	10000	F80		
18F4680	10000	D00		
18F4682	14000	D00		
18F4685	18000	D00		
18F46J11	FFF8	EC0		
18F46J13	FFF8	EB0		
18F46J50	FFF8	EC0		
18F46J53	FFF8	EB0		
18F46K20	10000	F60		
18F46K22	10000	F38		
18F46K80	10000	E41		
18F47J13	1FFF8	EB0		
18F47J53	1FFF8	EB0		
18F6310	2000	300		2000-1FFFFF
18F6390	2000	300		2000-1FFFFF
18F6393	2000	300		2000-1FFFFF
18F63J11	1FF8	400		20000-1FFFFF
18F63J90	1FF8	400		20000-1FFFFF
18F6410	4000	300		2000-1FFFFF
18F6490	4000	300		2000-1FFFFF
18F6493	4000	300		2000-1FFFFF
18F64J11	3FF8	400		20000-1FFFFF
18F64J90	3FF8	400		20000-1FFFFF
18F6520	8000	800		
18F6525	C000	F00		
18F6527	C000	F60		
18F6585	C000	D00		
18F65J10	7FF8	800		20000-1FFFFF
18F65J11	7FF8	800		20000-1FFFFF
18F65J15	BFF8	800		20000-1FFFFF
18F65J50	7FF8	F40		8000-1FFFFF
18F65J90	7FF8	800		20000-1FFFFF
18F65K22	8000	800		
18F65K80	8000	E41		
18F65K90	8000	800		
18F6620	10000	F00		
18F6621	10000	F00		
18F6622	10000	F60		
18F6627	18000	F60		
18F6628	18000	F60		
18F6680	10000	D00		
18F66J10	FFF8	800		20000-1FFFFF
18F66J11	FFF8	F40		10000-1FFFFF
18F66J15	17FF8	F60		20000-1FFFFF
18F66J16	17FF8	F40		18000-1FFFFF

continued...

Table C.1: Devices supported by HI-TECH C Compiler for PIC18 MCUs

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F66J50	FFF8	F40		10000-1FFFFF
18F66J55	17FF8	F40		18000-1FFFFF
18F66J60	FFF8	EE0		20000-1FFFFF
18F66J65	17FF8	EE0		20000-1FFFFF
18F66J90	FFF8	F54		20000-1FFFFF
18F66J93	FFF8	F54		20000-1FFFFF
18F66K22	10000	F16		
18F66K80	10000	E41		
18F66K90	10000	EF4		
18F6720	20000	F00		
18F6722	20000	F60		
18F6723	20000	F60		
18F67J10	1FFF8	F60		20000-1FFFFF
18F67J11	1FFF8	F40		20000-1FFFFF
18F67J50	1FFF8	F40		20000-1FFFFF
18F67J60	1FFF8	EE0		20000-1FFFFF
18F67J90	1FFF8	F54		20000-1FFFFF
18F67J93	1FFF8	F54		20000-1FFFFF
18F67K22	20000	F16		
18F67K90	20000	EF4		
18F8310	2000	300		1000-1FFFFF
18F8390	2000	300		2000-1FFFFF
18F8393	2000	300		2000-1FFFFF
18F83J11	1FF8	400		20000-1FFFFF
18F83J90	1FF8	400		20000-1FFFFF
18F8410	4000	300		2000-1FFFFF
18F8490	4000	300		2000-1FFFFF
18F8493	4000	300		2000-1FFFFF
18F84J11	3FF8	400		20000-1FFFFF
18F84J90	3FF8	400		20000-1FFFFF
18F8520	8000	800		8000-1FFFFF
18F8525	C000	F00		C000-1FFFFF
18F8527	C000	F60		C000-1FFFFF
18F8585	C000	D00		C000-1FFFFF
18F85J10	7FF8	800		20000-1FFFFF
18F85J11	7FF8	800		20000-1FFFFF
18F85J15	BFF8	800		20000-1FFFFF
18F85J50	7FF8	F40		8000-1FFFFF
18F85J90	7FF8	800		20000-1FFFFF
18F85K22	8000	800		
18F85K90	8000	800		
18F8620	10000	F00		10000-1FFFFF
18F8621	10000	F00		10000-1FFFFF
18F8622	10000	F60		10000-1FFFFF
18F8627	18000	F60		18000-1FFFFF

continued...

Table C.1: Devices supported by HI-TECH C Compiler for PIC18 MCUs

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18F8628	18000	F60		18000-1FFFFF
18F8680	10000	D00		10000-1FFFFF
18F86J10	FFF8	800		20000-1FFFFF
18F86J11	FFF8	F40		10000-1FFFFF
18F86J15	17FF8	F60		20000-1FFFFF
18F86J16	17FF8	F40		18000-1FFFFF
18F86J50	FFF8	F40		10000-1FFFFF
18F86J55	17FF8	F40		18000-1FFFFF
18F86J60	FFF8	EE0		20000-1FFFFF
18F86J65	17FF8	EE0		20000-1FFFFF
18F86J72	FFF8	F54		20000-1FFFFF
18F86J90	FFF8	F54		20000-1FFFFF
18F86J93	FFF8	F54		20000-1FFFFF
18F86K22	10000	F16		
18F86K90	10000	EF4		
18F8720	20000	F00		20000-1FFFFF
18F8722	20000	F60		20000-1FFFFF
18F8723	20000	F60		20000-1FFFFF
18F87J10	1FFF8	F60		20000-1FFFFF
18F87J11	1FFF8	F40		20000-1FFFFF
18F87J50	1FFF8	F40		20000-1FFFFF
18F87J60	1FFF8	EE0		20000-1FFFFF
18F87J72	1FFF8	F54		20000-1FFFFF
18F87J90	1FFF8	F54		20000-1FFFFF
18F87J93	1FFF8	F54		20000-1FFFFF
18F87K22	20000	F16		
18F87K90	20000	EF4		
18F96J60	FFF8	EE0		20000-1FFFFF
18F96J65	17FF8	EE0		20000-1FFFFF
18F97J60	1FFF8	EE0		20000-1FFFFF
18LF13K22	2000	100		
18LF13K50	2000	200		
18LF14K22	4000	200		
18LF14K50	4000	300		
18LF23K22	2000	200		
18LF24J10	3FF8	400		20000-1FFFFF
18LF24J11	3FF8	EC0		
18LF24J50	3FF8	EC0		
18LF24K22	4000	300		
18LF25J10	7FF8	400		20000-1FFFFF
18LF25J11	7FF8	EC0		
18LF25J50	7FF8	EC0		
18LF25K22	8000	600		
18LF25K80	8000	E41		
18LF26J11	FFF8	EC0		

continued...

Table C.1: Devices supported by HI-TECH C Compiler for PIC18 MCUs

DEVICE	ROMSIZE	RAMSIZE	EEPROMSIZE	EXTMEM
18LF26J13	FFF8	EB0		
18LF26J50	FFF8	EC0		
18LF26J53	FFF8	EB0		
18LF26K22	10000	F38		
18LF26K80	10000	E41		
18LF27J13	1FFF8	EB0		
18LF27J53	1FFF8	EB0		
18LF43K22	2000	200		
18LF44J10	3FF8	400		20000-1FFFFF
18LF44J11	3FF8	EC0		
18LF44J50	3FF8	EC0		
18LF44K22	4000	300		
18LF45J10	7FF8	400		20000-1FFFFF
18LF45J11	7FF8	EC0		
18LF45J50	7FF8	EC0		
18LF45K22	8000	600		
18LF45K80	8000	E41		
18LF46J11	FFF8	EC0		
18LF46J13	FFF8	EB0		
18LF46J50	FFF8	EC0		
18LF46J53	FFF8	EB0		
18LF46K22	10000	F38		
18LF46K80	10000	E41		
18LF47J13	1FFF8	EB0		
18LF47J53	1FFF8	EB0		
18LF65K80	8000	E41		
18LF66K80	10000	E41		

Index

! macro quote character, 172
\ command file character, 23
. psect address symbol, 194
.as files, 24
.cmd files, 206
.crf files, 51, 155
.hex files, 25
.lib files, 204, 205
.lnk files, 197
.lst files, 49
.obj files, 156, 194, 205
.opt files, 155
.p1 files, 24
.pro files, 59
.sym files, 193, 196
/ psect address symbol, 194
;; comment suppression characters, 172
<> macro quote characters, 172
? character
    in assembly labels, 160
??nnnn type symbols, 161, 173
@ command file specifier, 23
#asm directive, 134
#define, 42
#include directive, 22
#pragma directives, 143
#undef, 48
$ character
    in assembly labels, 160
$ location counter symbol, 161
% macro argument prefix, 172
& assembly macro concatenation character, 172
_ character
    in assembly labels, 160
_EEPROMSIZE, 141
_ERRATA_TYPES, 142
_FLASH_ERASE_SIZE, 141
_FLASH_WRITE_SIZE, 141
_HTC_EDITION_, 140
_HTC_VER_MAJOR_, 141
_HTC_VER_MINOR_, 141
_HTC_VER_PATCH_, 141
_ICDROM_END, 141
_ICDROM_START, 141
MPC, 141
_PIC18, 141
_PLIB, 143
_RAMSIZE, 141
_ROMSIZE, 141
__Bxxxx type symbols, 152
__CONFIG macro, 87, 88, 228
__DATE__, 143
__EEPROM_DATA, 89
__EEPROM_DATA macro, 229
__FILE__, 142
__Hxxxx type symbols, 152
__IDLOC macro, 230
__LINE__, 142
__Lxxxx type symbols, 152
__MPLAB_ICD__, 141

__MPLAB_PICKIT2__, 141
__MPLAB_PICKIT3__, 141
__MPLAB_REALICE__, 141
__PICC18__, 141
__TIME__, 143
__serial0 label, 63
24-bit doubles, 52, 54
32-bit doubles, 52, 54

abs function, 233
abs PSECT flag, 167
absolute address, 157
absolute object files, 193, 194
absolute psects, 167, 168
absolute variables, 115, 157
    bits, 96
access bank, 103, 157
acos function, 234
additional memory ranges, 60, 61
addresses
    byte, 217
    link, 189, 194
    load, 189, 194
    word, 217
addressing unit, 167
ALIGN directive, 173
alignment
    within psects, 173
ANSI standard
    conformance, 64
    divergence from, 83
    implementation-defined behaviour, 83
argument area, 117
argument passing, 117
ASCII characters, 96
asctime function, 235
asin function, 237
asm() C directive, 134
assembler, 153
    controls, 176
    directives, 165
    options, 154
    pseudo-ops, 165
assembler control
    COND, 176
    EXPAND, 176
    INCLUDE, 178
    LIST, 178
    NOCOND, 178
    NOEXPAND, 178
    NOLIST, 179
    NOXREF, 179
    PAGE, 179
    SPACE, 179
    SUBTITLE, 179
    TITLE, 179
    XREF, 179
assembler directive
    ALIGN, 173
    DABS, 170
    DB, 170
    DS, 170
    DW, 170
    ELSE, 171
    ELSIF, 171
    END, 33, 165
    ENDIF, 171
    ENDM, 172
    EQU, 158, 169
    FNCALL, 171
    FNROOT, 171
    GLOBAL, 162, 165
    IF, 171
    IRP, 174
    IRPC, 174
    LOCAL, 161, 173
    MACRO, 158, 172
    ORG, 169

PROCESSOR, 156, 175
PSECT, 164, 167
REPT, 174
SET, 158, 169
SIGNAT, 150, 176
assembler files
preprocessing, 58
assembler listings, 49
assembler optimizer
debug information and, 156
enabling, 156
viewing output of, 155
assembler option
-A, 155
-C, 155
-Cchipinfo, 155
-E, 155
-Flength, 155
-H, 155
-I, 155
-Llistfile, 156
-O, 156
-Ooutfile, 156
-Twidth, 156
-V, 156
-X, 156
-processor, 156
assembler-generated symbols, 161
assembly, 153
accessing C variables from, 135
C prototypes for, 131
called from C code, 131
character constants, 160
character set, 158
comments, 158
conditional, 171
constants, 160
default radix, 160
delimiters, 159
embedding in C code, 131
expressions, 162
generating from C, 48
identifiers, 160
data typing, 161
in-line, 136
include files, 178
initializing
bytes, 170
words, 170
labels, 131, 158, 162
line numbers, 156
location counter, 161
multi-character constants, 160
operators, 164
psects for, 131
radix specifiers, 160
relative jumps, 161
relocatable expression, 164
repeating macros, 174
reserving memory, 170
special characters, 159
special comment strings, 159
statement format, 158
strings, 160
volatile locations, 159
assembly labels, 131, 158, 162
? character, 160
$ character, 160
_chacrter, 160
making globally accessable, 165
scope, 162, 165
assembly listings
blank lines, 179
disabling macro expansion, 178
enabling, 178
excluding conditional code, 178
expanding macros, 155, 176
generating, 156

hexadecimal constants, 155
including conditional code, 176
new page, 179
page length, 155
page width, 156
radix specification, 155
subtitles, 179
titles, 179
assembly macros, 172
! character, 172
% character, 172
& symbol, 172
concatenation of arguments, 172
quoting characters, 172
suppressing comments, 172
assert function, 238
atan function, 239
atan2 function, 240
atof function, 241
atoi function, 242
atol function, 243
auto switch type, 146
auto variable area, 117
auto variables, 110
Avocet symbol file, 197

banked access, 157
BANKMASK macro, 133, 158
banks
RAM banks, 103
BANKSEL directive, 133, 175
bankx qualifier, 102
base specifier, *see* radix specifier
bases
C source, 93
biased exponent, 98
big endian format, 218
binary constants
assembly, 160
C, 93
bit clear instruction, 88
Bit instructions, 88
bit manipulation macros, 88
bit PSECT flag, 167
bit set instruction, 88
bit types
absolute, 96
in assembly, 167
bit-addressable Registers, 96
bit-fields, 99
initializing, 100
unamed, 100
blocks, *see* psects
bootloader, 61, 214, 222
bootloaders, 62
bsearch function, 244
bss psect, 125, 188
clearing, 188
byte addresses, 217

C standard libraries, 29, 30
ceil function, 246
cgets function, 247
char types, 96
character constants, 94
assembly, 160
checksum endianism, 50, 218
checksum psect, 124
checksum specifications, 207
checksums, 50, 214, 218
algorithms, 50, 218
endianism, 50, 218
chipinfo files, 155
cinit psect, 124
class PSECT flag, 167
classes, 191
address ranges, 191
boundary argument, 196

upper address limit, 196
clib suboption, 29
clrwdt instruction, 86
CLRWDT macro, 249
COD file, 58
code protection fuses, 86
command files, 23
command line driver, 21
command lines
HLINK, long command lines, 197
long, 23, 206
verbose option, 49
common access, 157
compiler errors
format, 38
compiler generated psects, 123
compiler-generate input files, 28
compiling
to assembly file, 48
to object file, 42
COND assembler control, 176
conditional assembly, 171
config psect, 124
config_read() function, 250
config_write() function, 250
configuration
word, 124
configuration fuses, 86
console I/O functions, 152
const psect, 124, 125
const qualifier, 101
constants
assembly, 160
C specifiers, 93
character, 94
string, *see* string literals
context retrieval, 128
context saving, 127
copyright notice, 48
cos function, 252
cosh function, 253
cputs function, 254
creating
libraries, 205
creating new, 123
CREF application, 155, 209
CREF option
-Fprefix, 209
-Hheading, 210
-Llen, 210
-Ooutfile, 210
-Pwidth, 210
-Sstoplist, 210
-Xprefix, 210
CREF options, 209
cromwell application, 211
cromwell option
-B, 214
-C, 213
-D, 213
-E, 213
-F, 213
-Ikey, 213
-L, 213
-M, 214
-N, 211
-Okey, 213
-P, 211
-V, 214
cromwell options, 211
cross reference
disabling, 179
generating, 209
list utility, 209
cross reference file, 155
generation, 155
cross reference listings, 51
excluding header symbols, 209

excluding symbols, 210
headers, 210
output name, 210
page length, 210
page width, 210
cross referencing
enabling, 179
cstack psect, 126
ctime function, 255

DABS directive, 170
data psect, 126, 188
copying, 189
data psects, 32
data types, 93
16-bit integer, 96
24-bit integer, 97
8-bit integer, 96
assembly, 161
char, 96
floating point, 98
int, 96
short, 96
short long, 97
DB directive, 170
debug information, 43
assembler, 156
optimizers and, 156
default psect, 165
default radix
assembly, 160
delta PSECT flag, 167
delta psect flag, 191
dependencies, 63
dependency checking, 26
destination register, 157
device selection, 50
device_id_read() function, 231, 232, 256
DI macro, 258
directives
asm, C, 134
assembler, 165
EQU, 162
div function, 260
divide by zero
result of, 121
doprnt.c source file, 34
doprnt.pre, 35
double type, 52, 54
driver
command file, 23
command format, 22
input files, 22
long command lines, 23
options, 22
single step compilation, 25
driver option
–CODEOFFSET, 51
–DOUBLE=type, 52, 54
–ERRATA=type, 53
–ERRFORMAT=format, 53
–ERRORS=number, 53
–LANG=language, 55
–MSGFORMAT=format, 53
–NODEL, 25
–OUTPUT=type, 58
–PASS1, 24, 27
–PRE, 27
–RUNTIME, 29
–RUNTIME=type, 31–33, 63
–WARN=level, 65
–WARNFORMAT=format, 53
-C, 27, 42
-Efile, 43
-G, 43
-I, 44
-L, 44, 45
-M, 47

-O, 35
-S, 48
driver options
–WARNFORMAT=format, 65
DS directive, 170
DW directive, 170

EEPROM Data, 89
EEPROM data, 124
eeprom memory
initializing, 89
reading, 90
writing, 90
eeprom qualifier, 89
eeprom variables, 89
eeprom_data psect, 124
EEPROM_READ, 90
eeprom_read function, 261
EEPROM_WRITE, 90
eeprom_write function, 261
EI macro, 258
ELSE directive, 171
ELSIF directive, 171
embedding serial numbers, 223
END directive, 33, 165
end_init psect, 124
endasm directive, 134
ENDIF directive, 171
ENDM directive, 172
enhanced symbol files, 193
environment variable
HTC_ERR_FORMAT, 38
HTC_MSG_FORMAT, 38
HTC_WARN_FORMAT, 38
EQU directive, 158, 162, 169
equating assembly symbols, 169
errata workarounds, 30, 53
error files
creating, 192
error messages, 43
formatting, 38
LIBR, 207
eval_poly function, 263
exceptions, 126
exp function, 264
EXPAND assembler control, 176
exponent, 98
expressions
assemby, 162
relocatable, 164
External memory interface, 52
external program space, 103
external variables, 103

fabs function, 265
far keyword, 103
far variables, 103
fast interrupt save/restore, 127
fast interrupts, 127
file extensions, 22
file formats
assembler listing, 49
Avocet symbol, 197
command, 206
creating with cromwell, 211
cross reference, 155, 209
cross reference listings, 51
dependency, 63
DOS executable, 194
enhanced symbol, 193
library, 204, 205
link, 197
object, 42, 194, 205
preprocessor, 58
prototype, 59
specifying, 58
symbol, 193
TOS executable, 194

files
intermediate, 56–58
output, 57
temporary, 56, 57
fill memory, 214
filling unused memory, 53, 218
Flash and EEPROM Libraries, 30
floating point data types, 98
biased exponent, 98
exponent, 98
format, 98
mantissa, 98
floating suffix, 94
floor function, 268
fmod function, 267
FNCALL directive, 171
FNROOT directive, 171
frexp function, 269
function
return values, 119
structures, 119
function pointers, 109
function prototypes, 151, 176
function return values, 119
function signatures, 176
functions
argument passing, 117
getch, 152
interrupt, 126
interrupt qualifier, 126
kbhit, 152
putch, 152
recursion, 83
return values, 119
returning from, 126
signatures, 150
written in assembler, 131

getch function, 152, 270
getchar function, 271
getche function, 270
gets function, 272
GLOBAL directive, 162, 165
global optimization, 57
global PSECT flag, 167
global symbols, 188
gmtime function, 273

hardware
initialization, 33
header files
htc.h, 96
problems in, 64
HEX file format, 221
HEX file map, 222
hex files
address alignment, 62
address map, 214
calculating check sums, 214
converting to other Intel formats, 214
data record, 62, 217
detecting instruction sequences, 214
embedding serial numbers, 215
extended address record, 222
filling unused memory, 53, 214
find and replacing instructions, 214
merging multiple, 214
multiple, 192
record length, 62, 214, 221
hexadecimal constants
assembly, 160
hexmate application, 25, 214
hexmate option
+prefix, 217
-CK, 218
-FILL, 218, 222
-FIND, 220
-FIND...,DELETE, 221

-FIND...,REPLACE, 221
-FORMAT, 221
-HELP, 222
-LOGFILE, 222
-MASK, 223
-O, 223
-SERIAL, 63, 223
-SIZE, 224
-STRING, 224
-STRPACK, 225
-addressing, 217
-break, 217
file specifications, 215
hexmate options, 215
HI-TIDE, 55
HI_TECH_C, 140
high priority interrupts, 126
htc.h, 96
HTC_ERR_FORMAT, 38
HTC_MSG_FORMAT, 38
HTC_WARN_FORMAT, 38

I/O
console I/O functions, 152
serial, 152
STDIO, 152
ICD support, 127
ID Locations, 88
ID locations, 124
idata psect, 62, 124
identifiers
assembly, 160
IDLOC, 88
idloc psect, 124
idloc_read() function, 275
idloc_write() function, 275
IEEE floating point format, 98
IF directive, 171
Implementation-defined behaviour, 83
division and modulus, 121
shifts, 121
in-line assembly, 127, 136
INCLUDE assembler control, 178
include files
assembly, 178
incremental builds, 26
INHX32, 214, 222
INHX8M, 214, 222
init psect, 125
initialization of variables, 32
inline pragma directive, 147
input files, 22
int data types, 96
intcode psect, 125
intcodelo psect, 125
integer suffix
long, 93
unsigned, 93
integral constants, 93
integral promotion, 120
Intermediate files, 58
intermediate files, 22, 26, 57
interrupt functions, 126
calling functions from, 127
context retrieval, 128
context saving, 127
returning from, 126
interrupt keyword, 126
interrupt priority, 126
interrupt service routines, 126
interrupts
configuring priorities, 130
fast, 127
handling in C, 126
priority of, 126
use of shadow registers, 127
Interrupts fast, 127
IRP directive, 174

IRPC directive, 174
isalnum function, 277
isalpha function, 277
isatty function, 279
isdigit function, 277
islower function, 277
itoa function, 280

kbhit function, 152
keyword
    auto, 110
    bankx, 102
    far, 103
    interrupt, 126
    low_priority, 126
    near, 102, 103
    persistent, 102
keywords
    disabling non-ANSI, 64

l.obj output file, 25
label field, 158
labels
    assembly, 131, 158, 162
    local, 173
labs function, 281
language support, 37
ldexp function, 282
ldiv function, 283
LFSR instruction, 136, 158
LIBR, 204
    command line arguments, 204
    error messages, 207
    listing format, 206
    long command lines, 206
    module order, 206
librarian, 204
    command files, 206
    command line arguments, 204, 206
    error messages, 207
    listing format, 206
    long command lines, 206
    module order, 206
libraries
    adding files to, 205
    creating, 205
    deleting files from, 205
    EEPROM, 30
    excluding, 62
    flash, 30
    flash and eeprom library naming convention, 30
    format of, 204
    linking, 196
    listing modules in, 205
    Microchip Compatible Peripheral Libraries, 30, 62
    module order, 206
    scanning additional, 44
    used in executable, 194
library
    difference between object file, 204
    manager, 204
library function
    __CONFIG, 228
    __EEPROM_DATA, 229
    __IDLOC, 230
    abs, 233
    acos, 234
    asctime, 235
    asin, 237
    assert, 238
    atan, 239
    atan2, 240
    atof, 241
    atoi, 242
    atol, 243
    bsearch, 244

ceil, 246
cgets, 247
config_read(), 250
config_write(), 250
cos, 252
cosh, 253
cputs, 254
ctime, 255
device_id_read(), 231, 232, 256
div, 260
eeprom_read, 261
eeprom_write, 261
eval_poly, 263
exp, 264
fabs, 265
floor, 268
fmod, 267
frexp, 269
getch, 270
getchar, 271
getche, 270
gets, 272
gmtime, 273
idloc_read(), 275
idloc_write(), 275
isalnum, 277
isalpha, 277
isatty, 279
isdigit, 277
islower, 277
itoa, 280
labs, 281
ldexp, 282
ldiv, 283
localtime, 284
log, 286
log10, 286
longjmp, 287
ltoa, 289
memcmp, 290
memmove, 292
mktime, 293
modf, 295
os_tsleep, 297
pow, 298
printf, 34, 299
putch, 302
putchar, 303
puts, 305
qsort, 306
rand, 308
readtimerX, 310
round, 312
scanf, 313
setjmp, 315
sin, 317
sinh, 253
sqrt, 319
srand, 320
strcat, 321
strchr, 322
strcmp, 324
strcpy, 326
strcspn, 327
strichr, 322
stricmp, 324
stristr, 338
strlen, 328
strncat, 329
strncmp, 331
strncpy, 333
strnicmp, 331
strpbrk, 335
strrchr, 336
strrichr, 336
strspn, 337
strstr, 338
strtod, 339

strtok, 343
strtol, 341
tan, 345
tanh, 253
time, 346
toascii, 348
tolower, 348
toupper, 348
trunc, 349
ungetc, 350, 351
ungetch, 352
utoa, 353
va_arg, 354
va_end, 354
va_start, 354
vscanf, 313
writetimerX, 356
xtoi, 357
library macro
CLRWDT, 249
DI, 258
EI, 258
NOP, 296
RESET, 311
SLEEP, 318
limit PSECT flag, 168
limiting number of error messages, 53
link addresses, 189, 194
linker, 187
command files, 197
command line arguments, 189, 197
invoking, 197
long command lines, 197
passes, 204
symbols handled, 188
linker defined symbols, 152
linker errors
aborting, 193
undefined symbols, 193
linker option
-Aclass=low-high, 191, 195
-Cpsect=class, 191
-Dsymfile, 191
-Eerrfile, 192
-F, 192
-Gspec, 192
-H+symfile, 193
-Hsymfile, 193
-I, 193
-Jerrcount, 193
-K, 193
-L, 193
-LM, 194
-Mmapfile, 194
-N, 194
-Nc, 194
-Ns, 194
-Ooutfile, 194
-Pspec, 194
-Qprocessor, 196
-Sclass=limit[,bound], 196
-Usymbol, 196
-Vavmap, 197
-Wnum, 197
-X, 197
-Z, 197
linker options, 189
adjusting use driver, 45
numbers in, 190
linking programs, 149
LIST assembler control, 178
list files
assembler, 49
little endian format, 93, 97, 218
load addresses, 189, 194
loadfsr, 158
LOCAL directive, 161, 173
local PSECT flag, 168

local psects, 188
local symbols, 49
    suppressing, 156, 197
local variables, 110
    auto, 110
    static, 114
localtime function, 284
location counter, 161, 169
log function, 286
LOG10 function, 286
long data types, 97
long integer suffix, 93
longjmp function, 287
low priority interrupts, 126
ltoa function, 289

MACRO directive, 158, 172
macros
    disabling in listing, 178
    expanding in listings, 155, 176
    nul operator, 172
    predefined, 140
    repeat with argument, 174
    undefining, 48
    unnamed, 174
main function, 27, 31
mantissa, 98
map files, 194
    generating, 47
    processor selection, 196
    segments, 199
    symbol tables in, 194
    width of, 197
maximum number of errors, 53
MDF, 36
memcmp function, 290
memmove function, 292
memory
    external program space, 103
    external RAM, 103
    reserving, 60, 61
    specifying, 60, 61
    specifying ranges, 191
    unused, 53, 194
memory pages, 168
memory summary, 64
merging hex files, 217
message
    language, 37
message description files, 36
messages
    disabling, 56
    warning, 56
Microchip COF file, 58
Microchip Compatible Peripheral Libraries, 30, 62
mixing C and assembly, 131
mktime function, 293
modf function, 295
module, 22
modules
    in library, 204
    list format, 206
    order in library, 206
    used in executable, 194
MOVFF instruction, 136
moving code, 51
MPLAB, 55
    build options, 45, 66, 73
    ICD support, 127
    plugin, 66
multi-character constants
    assembly, 160
multiple hex files, 192

near keyword, 102, 103
NOCOND assembler control, 178
NOEXPAND assembler control, 178

NOLIST assembler control, 179
non-volatile memory, 102
non-volatile RAM, 102
NOP macro, 296
NOXREF assembler control, 179
numbers
    C source, 93
    in linker options, 190
nv psect, 126
nvbit psect, 102
nvram, 102
nvram psect, 102
nvrram psect, 102

object code, version number, 194
object files, 42
    absolute, 193
    relocatable, 187
    specifying name of, 156
    suppressing local symbols, 156
    symbol only, 192
OBJTOHEX, 207
    command line arguments, 207
objtohex application, 25
offsetting code, 51
operators
    assembly, 164
Optimizations
    assembler, 57
    code generator, 57
    debugging, 57
    global, 57
optimizations
    assembler, *see* assembler optimizer
optimizing assembly code, 155
options
    assembler, 154
ORG directive, 169
os_tsleep function, 297

output
    specifying name of, 47
output directory, 57
output file, 47
output file formats, 194
    American Automation HEX, 58
    Binary, 58
    Bytecraft COD, 58
    COFF, 58
    ELF, 58
    Intel HEX, 58
    library, 58
    Microchip COFF, 58
    Motorola S19 HEX, 58
    specifying, 58, 207
    Tektronic, 58
    UBROF, 58
output files, 57
    l.obj, 25
    names of, 23
overlaid memory areas, 193
overlaid psects, 168
ovrld PSECT flag, 168

p-code files, 22
PAGE assembler control, 179
parameter passing, 117
passing parameters to assembly, 131
persistent keyword, 102
persistent qualifier, 102
PIC18 MCU assembly language, 157
PICC18
    predefined macros, 140
    supported data types, 93
PICC18, *see* driver
PICC18 options
    –EMI, 52
    –SUMMARY=type, 149
    -C, 149

-S, 149
PICC18 output formats
American Automation Hex, 35
Binary, 35
Bytecraft, 35
Intel Hex, 35
Motorola Hex, 35
Tektronix Hex, 35
UBROF, 35
PICC18 options
–CHAR=type, 96
pointer
qualifiers, 104
pointers, 104
16bit, 104
32 bit, 104
combining with type modifiers, 104
function, 109
to functions, 104
pow function, 298
powerup psect, 125
powerup routine, 31, 33
powerup.as, 33
pragma directives, 143
predefined symbols
preprocessor, 140
preprocessing, 48
assembler files, 48
preprocessor
macros, 42
path, 44
preprocessor directive
asm, 134
endasm, 134
preprocessor directives, 140
in assembly files, 158
preprocessor symbols
predefined, 140
printf
format checking, 143
printf function, 28, 299
printf_check pragma directive, 143
PROCESSOR directive, 156
processor ID data, 88
processor selection, 50, 175, 196
program entry point, 33, 165
program sections, 164
project name, 23
prototypes
for assembly code, 131
psect
bss, 125, 188
checksum, 124
cinit, 124
config, 124
const, 124, 125
cstack, 126
data, 126, 188
eeprom_data, 124
end_init, 124
idata, 62, 124
idloc, 124
init, 125
intcode, 125
intcodelo, 125
nv, 126
nvbit, 102
nvram, 102
nvrram, 102
powerup, 125
rbss, 62
text, 125
PSECT directive, 164, 167
PSECT directive flag
limit, 196
PSECT directive flags, 167
abs, 167
bit, 167

class, 167
delta, 167
global, 167
limit, 168
local, 168
ovrld, 168
pure, 168
reloc, 168
size, 168
space, 168
with, 168
psects, 123, 164, 188
absolute, 167, 168
aligning within, 173
alignment of, 168
basic kinds, 188
class, 191, 196
compiler generated, 123
default, 165
delta value of, 191
differentiating ROM and RAM, 168
for assembly code, 131
linking, 187
listing, 64
local, 188
maximum size of, 168
page boundaries and, 168
specifying address ranges, 195
specifying addresses, 191, 194
struct, 119
pseudo-ops
assembler, 165
pure PSECT flag, 168
putch function, 152, 302
putchar function, 303
puts function, 305

qsort function, 306
qualifier
auto, 110
bankx, 102
far, 103
near, 102
persistent, 102
volatile, 159
qualifiers, 101, 102
and auto variables, 110
const, 101
pointer, 104
volatile, 101
quiet mode, 48

radix specifiers
assembly, 160
binary, 93
C source, 93
decimal, 93
hexadecimal, 93
octal, 93
RAM access bit, 157
rand function, 308
rbss psect, 62
read-only variables, 101
READTIMERx function, 310
recursion, 83
redirecting errors, 43
reference, 190, 198
registers
shadow, 127
special function, *see* special function registers
regsused pragma directive, 145
relative jump, 161
RELOC, 192, 194
reloc PSECT flag, 168
relocatable
object files, 187
relocation, 187

relocation information
    preserving, 193, 194
REPT directive, 174
reserving memory, 60, 61
reset
    code executed after, 33
RESET macro, 311
RETFIE instruction, 126, 128, 157
RETLW instruction, 126
RETURN instruction, 126
return values, 119
rotate operation, 84
round function, 312
runtime environment, 63
RUNTIME option
    clear, 62
    clib, 62
    init, 62
    keep, 62
    no_startup, 62
    plib, 62
runtime startup
    variable initialization, 32
runtime startup code, 31
runtime startup module, 28, 62

scale value, 167
scanf function, 313
search path
    header files, 44
segment selector, 192
segments, *see* psects, 192, 199
serial I/O, 152
serial numbers, 63, 223
    accessing, 63
SET directive, 158, 169
setjmp function, 315
sfr.h, 158
SFRs
    multibyte, 92
shadow registers, 127
shift operations
    result of, 121
shifting code, 51
short long data types, 97
sign extension when shifting, 121
SIGNAT directive, 151, 176
signature checking, 150
signature values, 131
signatures, 176
sin function, 317
single step compilation, 25
sinh function, 253
size of doubles, 52, 54
size PSECT flag, 168
skipping applications, 64
SLEEP macro, 318
source file, 22
SPACE assembler control, 179
space PSECT flag, 168
space switch type, 146
special characters in assembly, 159
special function registers
    in assembly code, 162
    multibyte, 92
special type qualifiers, 102
speed switch type, 146
sports cars, 161
sqrt function, 319
srand function, 320
stack
    usage, 62
standard library files, 29, 30
standard type qualifiers, 101
start label, 33
startup module, 62
    clearing bss, 188
    data copying, 189

startup.as, 31
static variables, 114
STDIO, 152
storage class, 110
strcat function, 321
strchr function, 322
strcmp function, 324
strcpy function, 326
strcspn function, 327
strichr function, 322
stricmp function, 324
string literals, 94, 224
    concatenation, 94
String packing, 225
strings
    assembly, 160
    storage location, 94, 224
    type of, 94
stristr function, 338
strlen function, 328
strncat function, 329
strncmp function, 331
strncpy function, 333
strnicmp function, 331
strpbrk function, 335
strrchr function, 336
strrichr function, 336
strspn function, 337
strstr function, 338
strtod function, 339
strtok function, 343
strtol function, 341
struct psect, 119
structures
    bit-fields, 99
    qualifiers, 100
SUBTITLE assembler control, 179
SUMMARY option
    class, 65
    file, 65
    hex, 65
    mem, 65
    psect, 65
switch pragma directive, 146
symbol files, 43
    Avocet format, 197
    enhanced, 193
    generating, 193
    local symbols in, 197
    old style, 191
    removing local symbols from, 49
    removing symbols from, 196
    source level, 43
symbol tables, 194, 196
    sorting, 194
symbols
    assembler-generated, 161
    global, 188, 205
    linker defined, 152
    undefined, 196

table read instruction, 116
tan function, 345
tanh function, 253
temporary files, 57
text psect, 125
time function, 346
time switch type, 146
Timers, 310
timers function, 356
TITLE assembler control, 179
toascii function, 348
tolower function, 348
toupper function, 348
translation unit, 23
trunc function, 349
type checking
    assembly routines, 131

type modifiers
    combining with pointers, 104
type qualifier, 102
type qualifiers, 101
typographic conventions, 19

unamed structure members, 100
ungetc function, 350, 351
ungetch function, 352
universal toolsuite, 66
unnamed psect, 165
unsigned integer suffix, 93
unused memory
    filling, 214
utilities, 187
utoa function, 353

va_arg function, 354
va_end function, 354
va_start function, 354
variable initialization, 32
variables
    absolute, 115, 157
    accessing from assembly, 135
    auto, 110
    char types, 96
    floating point types, 98
    in external memory, 103
    int types, 96
    local, 110
    short long types, 97
    static, 114
verbose, 49
version number, 65
volatile qualifier, 101, 159
vscanf function, 313

W register, 159
warning level, 65
    setting, 197
warning message format, 65
warnings
    level displayed, 65
    suppressing, 197
with PSECT flag, 168
word addresses, 217
word boundaries, 168
writetimerx function, 356

XREF assembler control, 179
xtoi function, 357

PICC18 Command-line Options

Option	**Meaning**
`-C`	Compile to object files only
`-D`*`macro`*	Define preprocessor macro
`-E+`*`file`*	Redirect and optionally append errors to a file
`-G`*`file`*	Generate source-level debugging information
`-I`*`path`*	Specify a directory pathname for include files
`-L`*`library`*	Specify a library to be scanned by the linker
`-L-`*`option`*	Specify `-`*`option`* to be passed directly to the linker
`-M`*`file`*	Request generation of a MAP file
`-N`*`size`*	Specify identifier length
`-O`*`file`*	Output file name
`-P`	Preprocess assembler files
`-Q`	Specify quiet mode
`-S`	Compile to assembler source files only
`-U`*`symbol`*	Undefine a predefined preprocessor symbol
`-V`	Verbose: display compiler pass command lines
`-X`	Eliminate local symbols from symbol table
`--ADDRQUAL`	Set compiler response to memory qualifier
`--ASMLIST`	Generate assembler list file
`--CHECKSUM=`*`start-end@destination,type`*	Calculate a checksum
`--CHIP=`*`processor`*	Selects which processor to compile for
`--CHIPINFO`	Displays a list of supported processors
`--CMODE`	Specify compiler compatibility mode
`--CODEOFFSET=`*`address`*	Offset program code to address
`--CR=`*`file`*	Generate cross-reference listing
`--DEBUGGER=`*`type`*	Select the debugger that will be used
`--DOUBLE=size`	Selects size of `double` type
`--ECHO`	Echo command line
`--EMI=`*`type`*	Select mode of the external memory interface
`--ERRATA=`*`type`*	Add or remove specific software workarounds for silicon errata issues.
`--ERRFORMAT<=`*`format`*`>`	Format error message strings to the given style
`--ERRORS=`*`number`*	Sets the maximun number of errors displayed
`--FILL`	Specify fill value for unused program memory
`--FLOAT=`*`size`*	Selects size of `float` type
continued...	

PICC18 Command-line Options

Option	**Meaning**
`--GETOPTION=`*`app,file`*	Get the command line options for the named application
`--HELP<=option>`	Display the compiler's command line options
`--HTML`	Generate HTML debug files
`--IDE=`*`ide`*	Configure the compiler for use by the named IDE
`--LANG=`*`language`*	Specify language for compiler messages
`--MEMMAP=`*`file`*	Display memory summary information for the map file
`--MODE=`*`mode`*	Choose compiler operating mode
`--MSGDISABLE=`*`messagelist`*	Disable Warning Messages
`--MSGFORMAT<=`*`format`*`>`	Format general message strings to the given style
`--NODEL`	Do not remove temporary files generated by the compiler
`--NOEXEC`	Go through the motions of compiling without actually compiling
`--OBJDIR=`*`path`*	Specify intermediate files' directory
`--OPT<=`*`type`*`>`	Enable general compiler optimizations
`--OUTDIR=`*`path`*	Specify output files directory
`--OUTPUT=`*`type`*	Generate output file type
`--PRE`	Produce preprocessed source files
`--PROTO`	Generate function prototype information
`--RAM=`*`lo-hi`*`<,`*`lo-hi`*`,...>`	Specify and/or reserve RAM ranges
`--ROM=`*`lo-hi`*`<,`*`lo-hi`*`,...>`	Specify and/or reserve ROM ranges
`--RUNTIME=`*`type`*	Configure the C runtime libraries to the specified type
`--SCANDEP`	Generate file dependency ".DEP files"
`--SERIAL=`*`hexcode@address`*	Store a value in program memory
`--SETOPTION=`*`app,file`*	Set the command line options for the named application
`--SETUP=argument`	Setup the product
`--SHROUD`	Obfusate p-code files
`--STRICT`	Enable strict ANSI keyword conformance
`--SUMMARY=`*`type`*	Selects the type of memory summary output
`--TIME`	Report compilation times
`--VER`	Display the compiler's version number
`--WARN=`*`level`*	Set the compiler's warning level
continued...	

PICC18 Command-line Options

Option	**Meaning**
`--WARNFORMAT=`*`format`*	Format warning message strings to given style

ENROO 英锐恩

单片机集成方案全方位解决服务商

优质智能电子产品“芯”方案解决商

大中华地区代理商、PARTNER联系信息：

深圳市英锐恩科技有限公司
ENROO-TECH（SHENZHEN）CO.,LTD
中国·深圳市福田区福华路嘉汇新城汇商中心27楼2701

Enroo-Tech Technologies CO., Limited
Light-Tech International Limited
香港新界荃灣沙咀道 29-35 號科技中心 5 樓 5 室

联系电话：86-755-82543411,83167411,83283911,61357155, 88845951
联系传真：86-755-82543511
联系邮件：enroo@enroo.com
公司网站：http://www.enroo.com http://www.enroo-tech.com

英锐恩为 MICROCHIP 授权 PARTNER；代理销售全系列 MICROCHIP 单片机及模拟器件、接口电路。我们时刻为您提供技术支持;提供开发应用协助;提供成熟方案;提供免费样品;提供详细资料。

RECOGNIZED THIRD-PARTY TOOL PROVIDER
MICROCHIP

MICROCHIP AUTHORIZED DESIGN PARTNER